Diseño de Algoritmos en Pseudocódigo y Ordinogramas



{ Carlos Pes }

Libro del Tutorial de Algoritmos
www.abrirllave.com/algoritmos/

Primera edición, mayo 2017.

Dedicado a todos los profesores y profesoras de informática.

ÍNDICE DEL CONTENIDO

PREFACIO

Objetivo del libro

En programación, los tipos de datos, las variables, las constantes, los operadores, las expresiones y las instrucciones, son los elementos básicos que se pueden utilizar para diseñar algoritmos. Así pues, en este libro se estudia de qué manera se interrelacionan dichos elementos entre sí.

Véase en el tutorial de programación de Abrirllave que los algoritmos son utilizados en la fase de diseño de un programa.

Contenidos del libro

Este libro está pensado, fundamentalmente, para todos aquellos que quieran aprender a diseñar algoritmos utilizando pseudocódigo, así como, diagramas de flujo (ordinogramas).

Los contenidos de cada capítulo del libro se apoyan en las explicaciones de los anteriores. Se pretende de esta forma que el lector pueda adquirir conocimientos, gradualmente, empezando desde cero, y alcanzar la destreza necesaria para poner en práctica los principios básicos de la programación estructurada:

- Aplicación del diseño modular.
- Utilización, exclusivamente, de estructuras secuenciales, alternativas y repetitivas.
- Empleo de estructuras de datos adecuadas para manipular información.

Los capítulos del libro son:

- Capítulo 1: Qué es un algoritmo
- Capítulo 2: Introducción a los tipos de datos
- Capítulo 3: Identificadores, variables y constantes
- Capítulo 4: Tipos de datos definidos por el programador
- Capítulo 5: Operadores y expresiones
- Capítulo 6: Instrucciones primitivas
- Capítulo 7: Estructura de un algoritmo en pseudocódigo
- Capítulo 8: Ordinogramas
- Capítulo 9: Instrucciones de control alternativas
- Capítulo 10: Instrucciones de control repetitivas
- Capítulo 11: Instrucciones de control de salto
- Capítulo 12: Llamadas a subalgoritmos

Los contenidos de los primeros diez capítulos están basados en contenidos incluidos en el libro *"Empezar de cero a programar en lenguaje C"*, el cual es del mismo autor.

Material extra

En la web del tutorial de algoritmos "www.abrirllave.com/algoritmos/" se proporcionan más recursos: teoría, ejemplos, ejercicios resueltos, etc.

Erratas

Para comunicar cualquier comentario, sugerencia o error detectado en el texto, puede hacerlo escribiendo un correo electrónico a:

correo@abrirllave.com

Todas las sugerencias serán atentidas lo antes posible. Gracias de antemano por colaborar en la mejora del contenido de este libro.

Agradecimientos

Gracias a todas las personas –familiares, amigos e incluso desconocidos– que tanto me inspiráis y motiváis –en mayor o menor medida– para escribir libros o contenidos educativos de informática en la Web; desde 2006 en www.carlospes.com y desde 2014 en www.abrirllave.com. ¡Gracias a todos!

Carlos Pes
Pamplona, mayo de 2017.

Twitter: @CarlosPes
Blog: http://carlospes.blogspot.com

Capítulo 1

Qué es un algoritmo

En programación, un **algoritmo** establece, de manera genérica e informal, la secuencia de pasos o acciones que resuelve un determinado problema informático.

Los algoritmos constituyen la documentación principal que se necesita para poder iniciar la fase de codificación de un programa y, para representarlos, se utiliza, fundamentalmente, dos tipos de notación: pseudocódigo y diagramas de flujo (ordinogramas). El diseño de un algoritmo es independiente del lenguaje que después se vaya a utilizar para codificarlo.

1.1. Pseudocódigo

El **pseudocódigo** es un lenguaje de programación algorítmico; es un lenguaje intermedio entre el lenguaje natural y cualquier lenguaje de programación específico, como son: C, FORTRAN, Pascal, etc. No existe una notación formal o estándar de pseudocódigo, sino que, cada programador puede utilizar la suya propia. Ahora bien, en los algoritmos de ejemplo de este tutorial, la mayoría de las palabras que se utilizan son una traducción literal –del inglés al castellano– de las palabras que se usan en lenguaje C para escribir las instrucciones de los programas. Por tanto, el pseudocódigo empleado en este tutorial es, mayormente, *"C En Español (**CEE**)"*. Se pretende de esta forma facilitar al estudiante la codificación posterior de los algoritmos de los ejemplos al lenguaje C. La codificación de un algoritmo consiste en traducirlo a un lenguaje de programación específico, C en nuestro caso.

EJEMPLO Si se desea crear un programa que calcule la suma de dos números enteros cualesquiera introducidos por el usuario y, después, muestre por pantalla el resultado obtenido:

```
Introduzca el primer número (entero): 3
Introduzca el segundo número (entero): 5
La suma es: 8
```

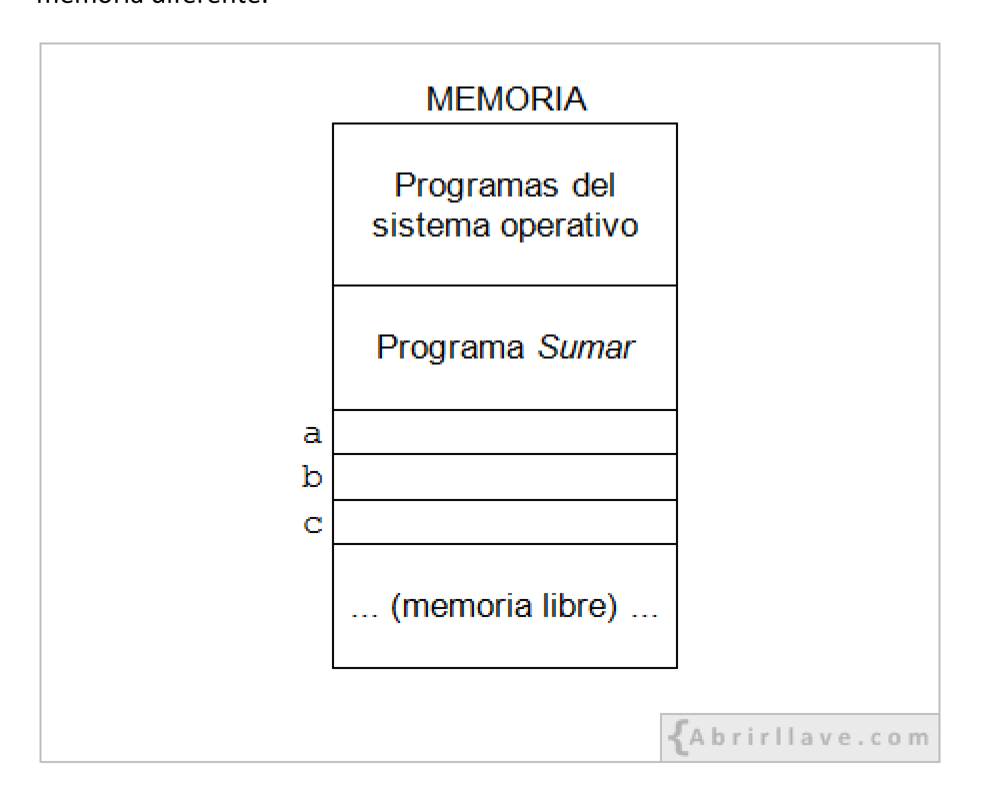
Se puede escribir el algoritmo siguiente:

```
algoritmo Sumar

variables
   entero a, b, c

inicio
   escribir( "Introduzca el primer número (entero): " )
   leer( a )
   escribir( "Introduzca el segundo número (entero): " )
   leer( b )
   c ← a + b
   escribir( "La suma es: ", c )
fin
```

Un algoritmo escrito en pseudocódigo siempre se suele organizar en tres secciones: cabecera, declaraciones y cuerpo. En la sección de cabecera se escribe el nombre del algoritmo, en este caso **Sumar**. En la sección de declaraciones se declaran algunos de los objetos que va a utilizar el programa. En el tutorial de lenguaje C de Abrirllave se estudian en detalle los distintos tipos de objetos que pueden ser utilizados en un programa, tales como: variables, constantes, subprogramas, etc. De momento, obsérvese que, en este ejemplo, las variables **a**, **b** y **c** indican que el programa necesita tres espacios en la memoria principal de la computadora para guardar tres números enteros. Cada una de las variables hace referencia a un espacio de memoria diferente.



En el cuerpo están descritas todas las acciones que se tienen que llevar a cabo en el programa, y siempre se escriben entre las palabras **inicio** y **fin**. La primera acción:

```
escribir( "Introduzca el primer número (entero): " )
```

Indica que se debe mostrar por pantalla el mensaje que hay entre comillas dobles. Después, mediante la acción:

```
leer( a )
```

Se está indicando que el programa esperará a que el usuario teclee un número entero, el cual se almacenará en el espacio de memoria representado por la variable **a**. El mismo proceso se tiene que seguir con el segundo número, que se guardará en el espacio de memoria representado por la variable **b**.

```
escribir( "Introduzca el segundo número (entero): " )
leer( b )
```

Acto seguido, la acción:

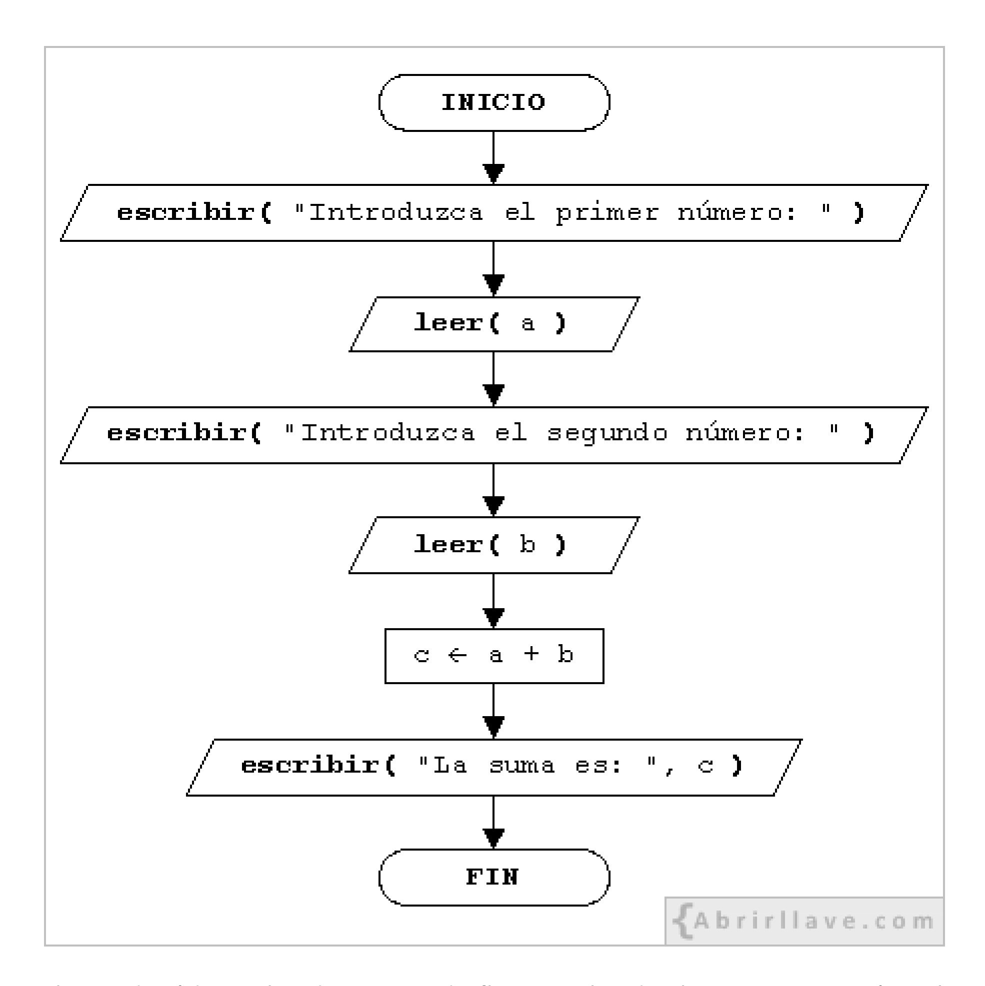
```
c ← a + b
```

Indica que en el espacio de memoria representado por la variable **c** se debe almacenar la suma de los dos números introducidos por el usuario del programa. Para terminar, el resultado de la suma se mostrará por pantalla con la acción:

```
escribir( "La suma es: ", c )
```

1.2. Diagramas de flujo (Ordinogramas)

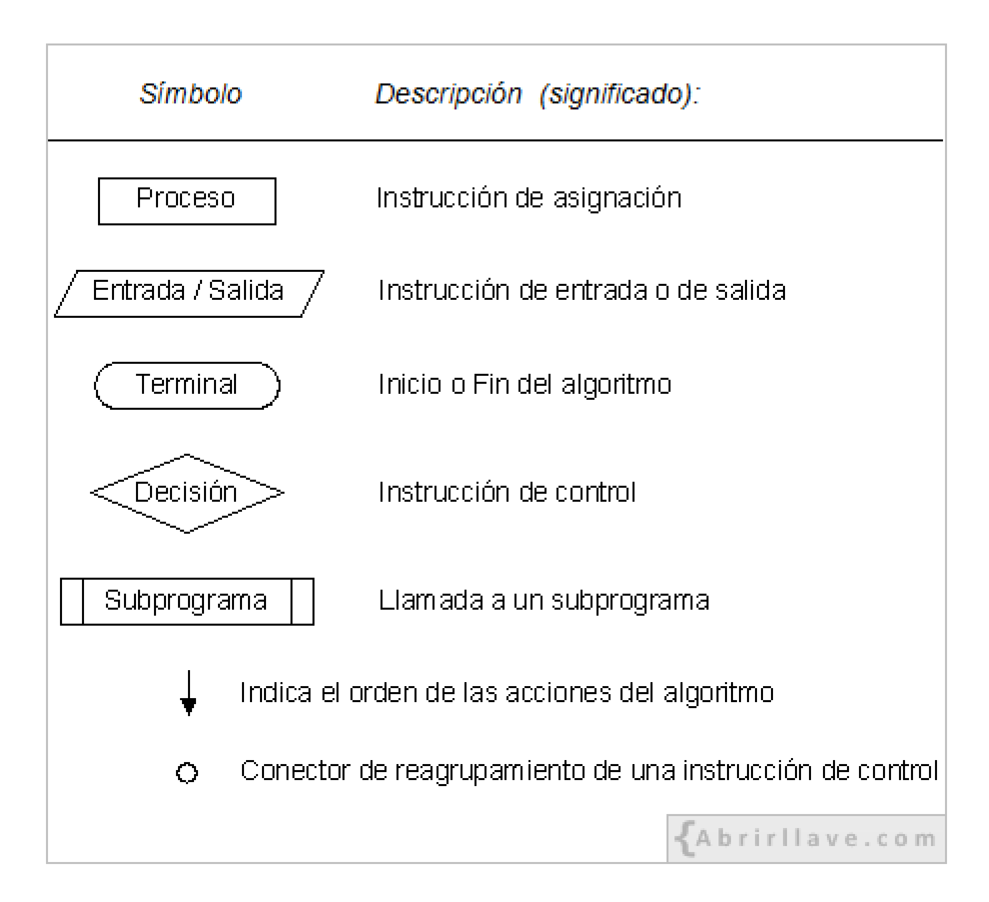
Los algoritmos también se pueden representar, gráficamente, por medio de **diagramas de flujo**. Los diagramas de flujo se pueden utilizar con otros fines, sin embargo, en este tutorial solamente los vamos a emplear para representar algoritmos. A tales diagramas de flujo también se les conoce como ordinogramas. Dicho de otra forma, un **ordinograma** representa, de manera gráfica, el orden de los pasos o acciones de un algoritmo. Por ejemplo, el algoritmo **`Sumar`** escrito en pseudocódigo en el apartado anterior se puede representar mediante el siguiente ordinograma:



El pseudocódigo y los diagramas de flujo son las dos herramientas más utilizadas para diseñar algoritmos en programación estructurada. Si bien, entre ambos tipos de representación existen las siguientes diferencias importantes:

- Los diagramas de flujo empezaron a utilizarse antes que el pseudocódigo.
- En pseudocódigo se suelen definir tres secciones del algoritmo (cabecera, declaraciones y cuerpo). Sin embargo, en un ordinograma únicamente se representa el cuerpo.
- En un ordinograma suele ser más fácil ver, a primera vista, cuál es el orden de las acciones del algoritmo.
- Los símbolos gráficos utilizados en un diagrama de flujo han sido estandarizados por el *American National Standards Institute* (**ANSI**). Sin embargo, no existe un "pseudocódigo estándar".

A continuación, se muestran los símbolos gráficos más utilizados para diseñar ordinogramas:



1.3. Cualidades de un algoritmo

Para cualquier problema dado no existe una única solución algorítmica; es tarea de la persona que diseña un algoritmo encontrar la solución más optima, ésta no es otra que aquella que cumple más fielmente las cualidades deseables de todo algoritmo bien diseñado:

- **Finitud**. Un algoritmo siempre tiene que finalizar tras un número finito de acciones. Cuando el algoritmo `Sumar` sea ya un programa, la ejecución de éste siempre será la misma, ya que, siempre se seguirán las acciones descritas en el cuerpo del algoritmo, una por una, desde la primera hasta la última y en el orden establecido.
- **Precisión**. Todas las acciones de un algoritmo deben estar bien definidas, esto es, ninguna acción puede ser ambigua, sino que cada una de ellas solamente se debe poder interpretar de una única manera. Dicho de otra forma, si el programa que resulta de un algoritmo se ejecuta varias veces con los mismos datos de entrada, en todos los casos se obtendrán los mismos datos de salida.
- **Claridad**. Lo normal es que un problema se pueda resolver de distintas formas. Por tanto, una de las tareas más importantes del diseñador de un algoritmo es encontrar la solución más legible, es decir, aquella más comprensible para el ser humano.
- **Generalidad**. Un algoritmo debe resolver problemas generales. Por ejemplo, el programa `Sumar` deberá servir para realizar sumas de dos números enteros cualesquiera, y no, solamente, para sumar dos números determinados, como pueden ser el 3 y el 5.

- **Eficiencia**. La ejecución del programa resultante de codificar un algoritmo deberá consumir lo menos posible los recursos disponibles del ordenador (memoria, tiempo de CPU, etc.).
- **Sencillez**. A veces, encontrar la solución algorítmica más eficiente a un problema puede llevar a escribir un algoritmo muy complejo, afectando a la claridad del mismo. Por tanto, hay que intentar que la solución sea sencilla, aun a costa de perder un poco de eficiencia, es decir, se tiene que buscar un equilibrio entre la claridad y la eficiencia. Escribir algoritmos sencillos, claros y eficientes se consigue a base de práctica.
- **Modularidad**. Nunca hay que olvidarse del hecho de que un algoritmo puede formar parte de la solución a un problema mayor. Pero, a su vez, dicho algoritmo debe descomponerse en otros, siempre y cuando, esto favorezca a la claridad del mismo.

La persona que diseña un algoritmo debe ser consciente de que todas las propiedades de un algoritmo se transmitirán al programa resultante.

1.4. Codificación

Una vez que los algoritmos de una aplicación han sido diseñados, ya se puede iniciar la fase de **codificación**. En esta etapa se tienen que traducir dichos algoritmos a un lenguaje de programación específico, en nuestro caso C; es decir, las acciones definidas en los algoritmos las vamos a convertir en instrucciones, también llamadas sentencias, del lenguaje C.

EJEMPLO Al codificar en C el algoritmo del programa **`Sumar`**, se escribirá algo parecido a:

```
#include <stdio.h>

int main()
{
   int a, b, c;

   printf( "\n   Introduzca el primer n%cmero (entero): ", 163 );
   scanf( "%d", &a );
   printf( "\n   Introduzca el segundo n%cmero (entero): ", 163 );
   scanf( "%d", &b );
   c = a + b;
   printf( "\n   La suma es: %d", c );

   return 0;
}
```

Para codificar un algoritmo hay que conocer la sintaxis del lenguaje al que se va a traducir. Sin embargo, independientemente del lenguaje de programación en que esté escrito un programa, será su algoritmo el que determine su lógica. La **lógica de un programa** establece cuáles son sus acciones y en qué orden se deben ejecutar. Por tanto, es conveniente que todo programador aprenda a diseñar algoritmos antes de pasar a la fase de codificación.

Capítulo 2

Tipos de datos

Los datos que utilizan los programas se pueden clasificar en base a diferentes criterios. Uno de los más significativos es aquel que dice que todos los datos que utilizan los programas son simples o compuestos.

Un **dato simple** es indivisible (atómico), es decir, no se puede descomponer.

EJEMPLO Un **año** es un dato simple.

```
Año...: 2006
```

Un **año** se expresa con un número entero, el cual no se puede descomponer. Sin embargo, un **dato compuesto** está formado por otros datos.

EJEMPLO Una **fecha** es un dato compuesto por tres datos simples (**día**, **mes**, **año**).

```
Fecha:
   Día...: 30
   Mes...: 11
   Año...: 2006
```

EJEMPLO Las **coordenadas** de un punto en un plano es también un dato compuesto, en este caso, por dos datos simples (**x**, **y**).

```
Coordenadas:
   X...: 34
   y...: 21
```

EJEMPLO Otro ejemplo de dato simple es una **letra**.

```
Letra...: t
```

Una **`letra`** se representa con un carácter del alfabeto. Pero, cuando varias letras se agrupan, entonces se obtiene un dato compuesto por varios caracteres.

EJEMPLO Para formar un **`nombre`** de persona se utilizan varios caracteres.

```
Nombre...: Ana
```

`Ana` es un dato compuesto por tres caracteres.

EJEMPLO Otro ejemplo de dato compuesto es una **`ficha`** que contenga el **`nombre`** de una persona, su **`ciudad`** de residencia y su **`fecha`** de nacimiento.

```
Ficha:
   Nombre...: Maite
   Ciudad...: Pamplona
   Fecha:
      Día...: 22
      Mes...: 4
      Año...: 1984
```

En este caso, la **`ficha`** es un dato compuesto por tres datos y, a su vez, todos ellos también son compuestos.

A los datos compuestos también se les conoce como **datos estructurados**, ya que, son datos que se forman al agruparse otros. Por consiguiente, de los datos simples se dice que no tienen estructura.

Seguidamente, se van a estudiar cinco tipos de datos:

- Entero
- Real
- Lógico
- Carácter
- Cadena

De ellos, tan solo el tipo cadena es compuesto. Los demás son los tipos de datos simples considerados **estándares**. Esto quiere decir que la mayoría de los lenguajes de programación permiten trabajar con ellos. Por ejemplo, en C es posible utilizar datos de tipo entero, real y carácter, sin embargo, los datos de tipo lógico no se pueden utilizar, ya que, no existen en este lenguaje.

Existen otros tipos de datos, simples y compuestos, que se estudiarán más adelante.

A los tipos de datos simples estándares también se les conoce como tipos de datos **primitivos**, **básicos** o **fundamentales**.

2.1. Datos de tipo numérico

Como su propio nombre indica, un **dato de tipo numérico** es aquel que puede tomar por valor un número. Existen dos tipos de datos numéricos básicos:

- Entero
- Real

EJEMPLO El número de **`asignaturas aprobadas`** por un estudiante en la universidad es un dato de tipo entero, mientras que, su **`nota`** en el examen de una asignatura en concreto puede ser de tipo real.

```
Asignaturas aprobadas.......: 4
Nota del examen de física...: 7,5
```

2.2. Datos de tipo entero

Un **dato de tipo entero** es aquel que puede tomar por valor un número perteneciente al conjunto de los números enteros (*Z*), el cual está formado por los números naturales, sus opuestos (números negativos) y el cero.

Z = { ..., -3, -2, -1, 0, 1, 2, 3, ... }

EJEMPLO La **edad** de una persona y el **año** en que nació, son dos datos de tipo entero.

```
Edad...: 29
Año....: 1976
```

Z es un conjunto infinito de números enteros, y como el ordenador no puede representarlos todos, un dato de tipo entero solamente puede tomar por valor un número perteneciente a un subconjunto de Z. Los valores máximo y mínimo de dicho subconjunto varían según las características de cada ordenador y del compilador que se utilice.

En pseudocódigo, para indicar que un dato es de tipo entero se utiliza la palabra reservada:

```
entero
```

En todos los lenguajes de programación existe un conjunto de palabras que tienen un significado especial, a estas palabras se las llama *reservadas*.

2.3. Datos de tipo real

Un **dato de tipo real** es aquel que puede tomar por valor un número perteneciente al conjunto de los números reales (*R*), el cual está formado por los números racionales e irracionales.

EJEMPLO El **`peso`** de una persona (en kilogramos) y su **`altura`** (en centímetros), son datos que pueden considerarse de tipo real.

```
Peso.....: 75,3
Altura...: 172,7
```

R es un conjunto infinito de números reales, y como el ordenador no puede representarlos todos, un dato de tipo real solamente puede tomar por valor un número perteneciente a un subconjunto de *R*. Los valores de dicho subconjunto varían según las características de cada ordenador y del compilador que se utilice.

En pseudocódigo, para indicar que un dato es de tipo real se utiliza la palabra reservada:

```
real
```

2.4. Datos de tipo lógico

Un **dato de tipo lógico** es aquel que puede tomar por valor únicamente uno de los dos siguientes:

```
{ verdadero, falso }
```

Los valores **`verdadero`** y **`falso`** son contrapuestos, de manera que, un dato de tipo lógico siempre está asociado a que algo se cumpla o no se cumpla.

EJEMPLO El **`estado`** de una barrera de paso de trenes es un dato que puede considerarse de tipo lógico, por ejemplo, asociando **`verdadero`** a que esté subida y **`falso`** a que esté bajada.

```
Estado...: falso
```

`falso` indica que la barrera está bajada.

En pseudocódigo, para indicar que un dato es de tipo lógico se utiliza la palabra reservada:

```
logico
```

A los datos de tipo lógico también se les conoce como **datos de tipo booleano** en nombre del matemático George Boole (1815-1864), que fue quien desarrolló el llamado *álgebra de Boole*, aplicado en informática en distintos ámbitos, tales como el diseño de ordenadores o la programación.

En C no existen los datos de tipo lógico. No obstante, se pueden simular con datos de tipo entero, considerándose el valor cero (0) como **`falso`**, y cualquier otro valor entero como **`verdadero`**.

2.5. Datos de tipo carácter

Un **dato de tipo carácter** es aquel que puede tomar por valor un carácter perteneciente al conjunto de los caracteres que puede representar el ordenador.

En pseudocódigo, el valor de un dato de tipo carácter se puede representar entre *comillas simples* (`'`) o *dobles* (`"`). Pero, en este tutorial, se van a utilizar solamente las comillas simples, al igual que se hace en C.

EJEMPLO En un examen con preguntas en las que hay que seleccionar la respuesta correcta entre varias opciones dadas (a, b, c, d, e), la **`respuesta correcta`** de cada una de las preguntas es un dato de tipo carácter.

```
Respuesta correcta a la pregunta 3...: 'c'
```

En pseudocódigo, para indicar que un dato es de tipo carácter se utiliza la palabra reservada:

```
caracter
```

2.6. Datos de tipo cadena

Un dato de tipo cadena es aquel que puede tomar por valor una secuencia de caracteres.

En pseudocódigo, el valor de un dato de tipo cadena se puede representar entre *comillas simples* (`'`) o *dobles* (`"`). Sin embargo, en este tutorial, se van a utilizar solamente las comillas dobles, al igual que se hace en C.

EJEMPLO El **`título`** de un libro y el **`nombre`** de su autor, son datos de tipo cadena.

```
Título...: "La Odisea"
Autor....: "Homero"
```

- **`"La Odisea"`** es una cadena de 9 caracteres.
- **`"Homero"`** es una cadena de 6 caracteres.

Fíjese que, en la cadena **`"La Odisea"`**, el carácter espacio en blanco también se cuenta.

En pseudocódigo, para indicar que un dato es de tipo cadena se utiliza la palabra reservada:

```
cadena
```

2.7. Clasificación de los tipos de datos simples

Los tipos de datos simples se clasifican en predefinidos y definidos por el programador. La clasificación completa es:

Tipos de datos simples (sin *estructura*) ***en pseudocódigo:***

Predefinidos (estándares):
Numéricos:
Entero (**entero**)
Real (**real**)
Lógico (**logico**)
Carácter (**caracter**)

Definidos por el programador (no *estándares*):
Subrangos (**subrango**)
Enumerados (**enumerado**)

Los **tipos de datos simples predefinidos** (estándares) son aquellos proporcionados por los lenguajes de programación. Pero, el programador también puede definir sus propios tipos de datos simples (subrangos y enumerados), los cuales se estudiarán más adelante.

Todos los datos simples son ordinales, excepto el dato de tipo real. Un **dato ordinal** es aquel que puede tomar por valor un elemento perteneciente a un conjunto en el que todo elemento tiene un predecesor y un sucesor, excepto el primero y el último. Por ejemplo, el valor 5, perteneciente al conjunto de los números enteros, tiene como predecesor al 4, y como sucesor al 6. Sin embargo, entre dos números reales siempre hay un número infinito de números.

Ejercicios resueltos
• Clasificar datos • Crucigrama de tipos de datos

Capítulo 3

Identificadores, variables y constantes

Para diseñar algoritmos en pseudocódigo, se pueden utilizar los siguientes elementos:

- Tipos de datos
- Variables
- Constantes
- Operadores
- Expresiones
- Instrucciones

Todos ellos están relacionados entre sí. En este capítulo se va a ver la relación que existe entre los tipos de datos, las variables y las constantes.

3.1. Identificadores

La mayoría de los elementos de un algoritmo escrito en pseudocódigo se diferencian entre sí por su nombre. Por ejemplo, los tipos de datos básicos se nombran como: **`entero`**, **`real`**, **`logico`** y **`caracter`**.

Cada uno de ellos es un identificador. Un **identificador** es el nombre que se le da a un elemento de un algoritmo (o programa). Por ejemplo, el tipo de dato **`entero`** hace referencia a un tipo de dato que es distinto a todos los demás tipos de datos, es decir, los valores que puede tomar un dato de tipo entero, no son los mismos que los que puede tomar un dato de otro tipo.

Los identificadores **`entero`**, **`real`**, **`logico`** y **`caracter`** están predefinidos, forman parte del lenguaje algorítmico. No obstante, en un algoritmo también pueden existir identificadores definidos por el programador. Por ejemplo, un algoritmo puede utilizar variables y constantes definidas por el programador. Además, los algoritmos también se deben nombrar mediante un identificador.

En pseudocódigo, a la hora de asignar un nombre a un elemento de un algoritmo, se debe de tener en cuenta que todo identificador debe cumplir unas reglas de sintaxis. Para ello, en

nuestro pseudocódigo CEE (*“C en Español”*), vamos a seguir las mismas reglas de sintaxis que existen en lenguaje C:

1. Consta de uno o más caracteres.
2. El primer carácter debe ser una letra o el carácter guion bajo (**_**), mientras que, todos los demás pueden ser letras, dígitos o el carácter guion bajo (**_**). Las letras pueden ser minúsculas o mayúsculas del alfabeto inglés. Así pues, no está permitido el uso de las letras ‘ñ’ y ‘Ñ’.
3. No pueden existir dos identificadores iguales, es decir, dos elementos de un algoritmo no pueden nombrarse de la misma forma. Lo cual no quiere decir que un identificador no pueda aparecer más de una vez en un algoritmo.

De la segunda regla se deduce que un identificador no puede contener caracteres especiales, salvo el carácter guion bajo (**_**). Es importante resaltar que las vocales no pueden llevar tilde ni diéresis.

EJEMPLO Algunos identificadores válidos que pueden ser definidos por el programador son:

```
numero

dia_del_mes

PINGUINO1

_ciudad

Z
```

EJEMPLO Los siguientes identificadores no son válidos por incumplir la segunda regla:

```
123

_DÍA

numero*

lugar de nacimiento

año
```

EJEMPLO Los siguientes identificadores no pueden ser definidos por el programador:

```
entero

logico
```

`entero` y **`logico`** son identificadores predefinidos (ya existen), por tanto, no pueden ser definidos por el programador, en cumplimiento de la tercera regla.

Los identificadores son sensibles a minúsculas y mayúsculas.

EJEMPLO **`Mes`** y **`mes`** son considerados identificadores distintos.

Es aconsejable que los identificadores tengan un significado afín a lo que representan.

EJEMPLO El algoritmo de un programa que tiene las tareas de:

1º) Recoger por teclado dos datos de tipo entero (como datos de entrada).

2º) Realizar la suma de ellos.

3º) Mostrar por pantalla el resultado de la suma (como dato de salida).

Estará mejor nombrado mediante el identificador **`SUMA`**, que, por ejemplo, mediante el identificador **`Algoritmo_1`**, aunque ambos identificadores sean válidos sintácticamente.

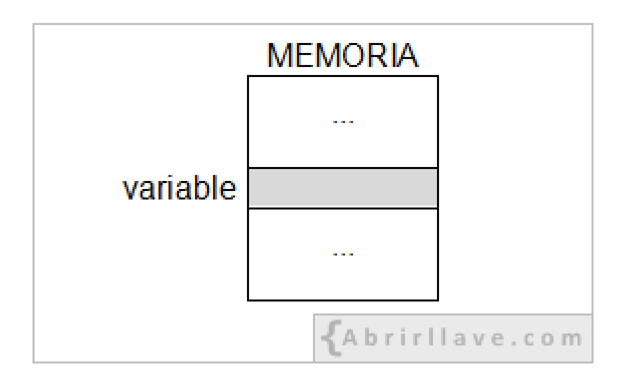
3.1.1. Palabras reservadas

Las **palabras reservadas** son identificadores predefinidos (tienen un significado especial). En todos los lenguajes de programación existe un conjunto de palabras reservadas. Por el momento, en pseudocódigo, se han estudiado las siguientes: **`cadena`**, **`caracter`**, **`entero`**, **`falso`**, **`logico`**, **`real`** y **`verdadero`**.

Ejercicio resuelto
• Identificadores válidos

3.2. Variables

Una **variable** representa a un espacio de memoria en el cual se puede almacenar un dato. Gráficamente, se puede representar como:



Como ya se ha estudiado, los datos que utilizan los programas pueden ser de diferentes tipos. Así que, de cada variable se debe especificar –definir– el tipo de dato que puede almacenar.

EJEMPLO Si un programa va a utilizar un dato de tipo carácter, será necesaria una variable de tipo carácter, y en el espacio de memoria reservado para dicha variable se podrá almacenar

cualquier carácter perteneciente al conjunto de los caracteres representables por el ordenador.

El programador, cuando desarrolla un programa –o diseña un algoritmo– debe decidir:

1. Cuántas son las variables que el programa necesita para realizar las tareas que se le han encomendado.
2. El tipo de dato que puede almacenar cada una de ellas.

Durante la ejecución de un programa, el valor que tome el dato almacenado en una variable puede cambiar tantas veces como sea necesario, pero, siempre, tomando valores pertenecientes al tipo de dato que el programador ha decidido que puede almacenar dicha variable, ya que, el tipo de dato de una variable no puede ser cambiado durante la ejecución de un programa.

3.2.1. Declaración de variables

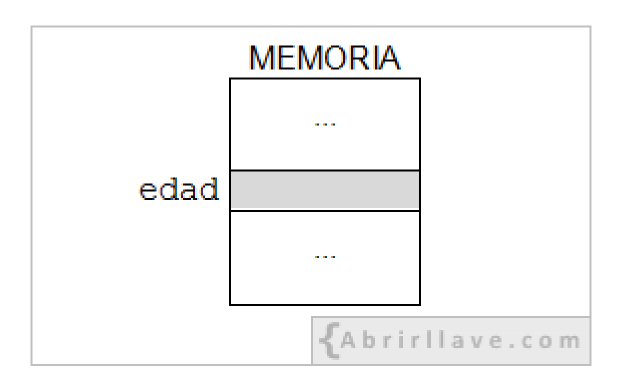
Para que un programa pueda hacer uso de una o más variables, estas deben ser declaradas previamente. Todas las variables de un programa se declaran de la misma forma, indicando de cada una de ellas:

- El tipo de dato que puede almacenar (mediante un identificador).
- Su nombre (mediante otro identificador).

EJEMPLO La declaración de una variable para almacenar la edad de una persona se escribe:

```
entero edad
```

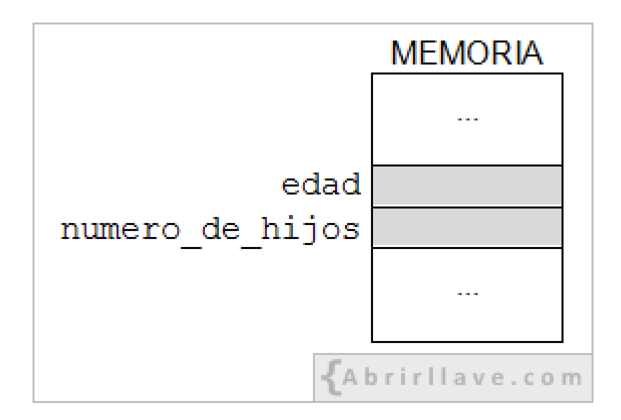
Por tanto, en la memoria de la computadora se reservará un espacio para almacenar la **edad**:



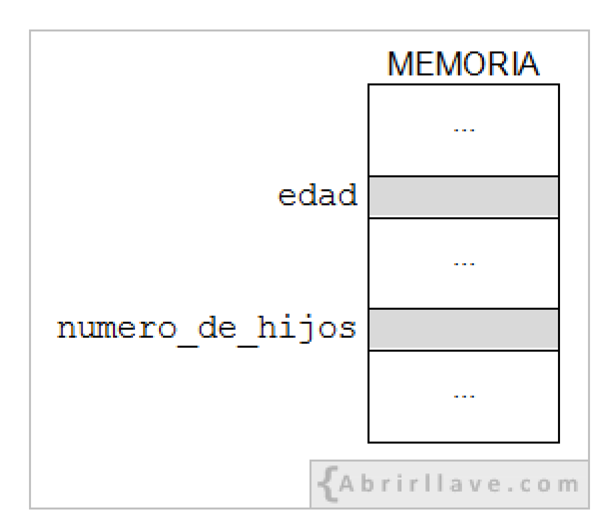
En un programa no se pueden declarar varias variables con el mismo nombre, salvo excepciones que estudiaremos más adelante. Sin embargo, sí pueden existir varias variables del mismo tipo de dato.

Siguiendo con el ejemplo anterior, si también se quiere declarar una variable para almacenar su número de hijos, se debe escribir:

```
entero edad
entero numero_de_hijos
```



Las variables de un programa no tienen por qué estar contiguas en la memoria del ordenador:



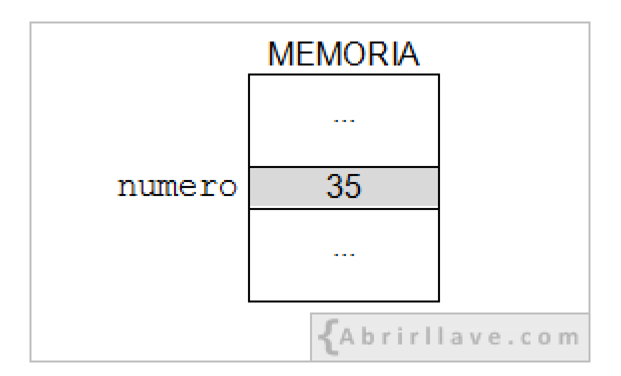
Puesto que las dos variables son del mismo tipo de dato, se pueden declarar en la misma línea separándolas por medio de una *coma* (,).

```
entero edad, numero_de_hijos
```

Opcionalmente, cuando se declara una variable, a esta se le puede asignar un valor inicial.

EJEMPLO Si se desea declarar una variable para almacenar un número entero y que, inicialmente, contenga el valor 35, se debe escribir:

```
entero numero = 35
```



Por consiguiente, para declarar una variable, en pseudocódigo utilizaremos la sintaxis:

```
<nombre_del_tipo_de_dato> <nombre_de_la_variable> [ = <expresión> ]
```

Y para declarar más de una variable del mismo tipo:

```
<nombre_del_tipo_de_dato> <nombre_de_la_variable_1> [ = <expresión_1> ],
                          <nombre_de_la_variable_2> [ = <expresión_2> ],
                            ...,
                          <nombre_de_la_variable_n> [ = <expresión_n> ]
```

Los caracteres *abrir corchete* (**[**) y *cerrar corchete* (**]**) se utilizan para indicar que lo que contienen es opcional.

Una **expresión** representa a un valor de un tipo de dato. En el ejemplo anterior, el valor 35 es de tipo entero. Más adelante se estudiarán en detalle las expresiones.

Durante la ejecución de un programa, para hacer uso del espacio de memoria representado por una variable, se utiliza su identificador.

Una variable puede ser declarada de cualquier tipo de dato (simple o compuesto). El tipo de dato de una variable determina su tamaño en memoria, o dicho de otro modo, establece el tamaño del espacio de memoria que se reserva para ella.

3.2.2. Símbolos reservados

Los **símbolos reservados** son aquellos caracteres que tienen un significado especial. En todos los lenguajes de programación existe un conjunto de símbolos reservados. Hasta ahora, solamente se han estudiado los símbolos utilizados en la sintaxis para declarar variables.

Símbolos reservados	
Símbolo	***Descripción***
=	Separador del identificador de una variable y de su expresión asignada en su declaración.
,	Separadora de los identificadores de varias variables en su declaración.

3.3. Constantes

Una **constante** representa a un valor –dato almacenado en memoria– que no puede cambiar durante la ejecución de un programa.

En lenguaje C, una constante puede ser de tipo entero, real, carácter, cadena o enumerado. Las constantes de tipo enumerado se van a estudiar en el capítulo siguiente. En cuanto a las demás, se pueden expresar de dos formas diferentes:

- Por su valor.
- Con un nombre (identificador).

EJEMPLO Las siguientes constantes de tipo entero están expresadas por su valor:

```
-5

10
```

Para expresar una constante con un nombre, la constante debe ser declarada previamente. Todas las constantes que se declaran en un programa son definidas de la misma forma, indicando de cada una de ellas:

- Su nombre (mediante un identificador).
- El valor que simboliza (mediante una expresión).

En pseudocódigo, para declarar una constante, vamos a utilizar la sintaxis:

```
<nombre_de_la_constante> = <expresión>
```

Y para declarar más de una constante en una misma línea, las separaremos por medio de comas (`,`).

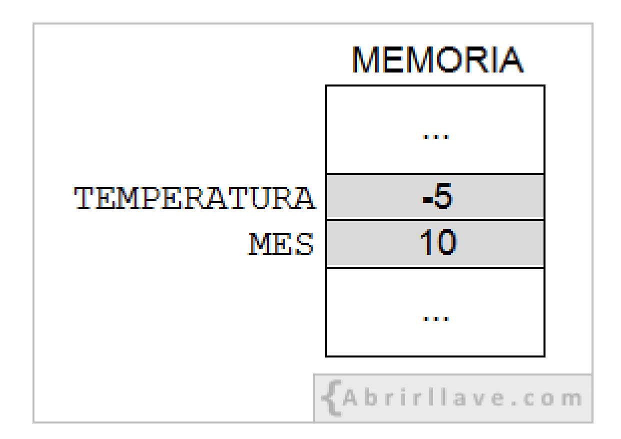
EJEMPLO De modo que, si se quieren declarar las constantes de tipo entero del ejemplo anterior, asignándoles un identificador, se puede escribir, por ejemplo:

```
TEMPERATURA = -5

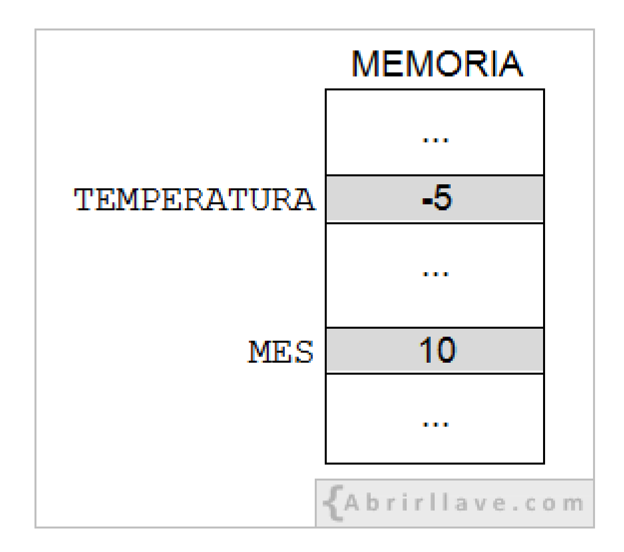
MES = 10
```

O también:

```
TEMPERATURA = -5, MES = 10
```



Al igual que las variables de un programa, tampoco las constantes tienen por qué estar contiguas en la memoria:



En programación es una buena práctica escribir los identificadores de las constantes en mayúsculas, de esta forma es más fácil localizarlos en el código de un programa (o algoritmo). Durante la ejecución de un programa, por medio del identificador de una constante, se puede hacer referencia al valor (dato) que simboliza, tantas veces como sea necesario.

Los símbolos reservados *igual* (**=**) y *coma* (**,**) han vuelto a aparecer.

Símbolos reservados	
Símbolo	***Descripción***
=	Separador del identificador de una constante y de su expresión asignada en su declaración.
,	Separadora de los identificadores de dos constantes en una misma línea.

3.3.1. Constantes de tipo entero

Una **constante de tipo entero** es aquella que representa a un valor –dato– perteneciente al subconjunto de *Z* representable por el ordenador.

EJEMPLO Suponiendo que el ordenador –utilizando dieciséis bits– pueda representar, en Complemento a 2, el siguiente conjunto de valores enteros:

{ -32768, -32767, ..., -1, 0, 1, ..., 32766, 32767 }

Algunos ejemplos de constantes de tipo entero son:

```
-32000

0

000077

+1111
```

Obsérvese que, además de los caracteres numéricos, dígitos del (**0**) al (**9**), también se puede hacer uso de los caracteres especiales (**+**) y (**−**) para indicar el signo de un número entero, el cual es positivo por omisión. Sin embargo, en pseudocódigo, y también en lenguaje C, es incorrecto usar los caracteres coma (**,**) y/o punto (**.**) para expresar constantes de tipo entero.

EJEMPLO Por tanto, es incorrecto escribir:

```
-32.000

0,0

+1,111.00
```

EJEMPLO Otros ejemplos incorrectos de constantes de tipo entero son:

++111 (No se puede duplicar el signo).

38000 (No pertenece al subconjunto de *Z* representable por el ordenador).

Han aparecido dos nuevos símbolos reservados.

Símbolos reservados	
Símbolo	***Descripción***
+	Indica que el número es positivo (es opcional).
−	Indica que el número es negativo.

3.3.2. Constantes de tipo real

Una **constante de tipo real** es aquella que representa a un valor –dato– perteneciente al subconjunto de *R* representable por el ordenador.

EJEMPLO Algunos ejemplos son:

`8.12`

`000.333` (Los ceros a la izquierda no son significativos)

`+1111.809`

`-3200.` (También se puede escribir `-3200.0`)

`.56` (También se puede escribir `0.56`)

Obsérvese que, además de los caracteres numéricos, dígitos del (**0**) al (**9**), también se puede hacer uso de los caracteres especiales (**+**) y (**-**) para indicar el signo de un número real. Además, en lenguaje C y, por tanto, también en nuestro pseudocódigo CEE, obligatoriamente debe aparecer el carácter punto (**.**), o el carácter (**e**) o (**E**) seguido del exponente, del cual también puede indicarse su signo con los caracteres (**+**) y (**-**). Los signos del exponente y del número en sí, por omisión, son positivos.

EJEMPLO Las siguientes constantes de tipo real están expresadas correctamente:

```
-77e-3

+1111e+2

2000E+2

3040e2
```

Una constante de tipo real también se puede expresar con el carácter punto (**.**) y el exponente al mismo tiempo.

EJEMPLO Algunos ejemplos son:

```
-50.50e-4

400.e-3

+65.65E+2

.7e3
```

El exponente tiene la función de desplazar la posición del punto decimal hacia la derecha si es positivo, o hacia la izquierda si es negativo.

EJEMPLO Así pues, las siguientes constantes de tipo real representan al mismo valor:

```
0.004E+3

4.

.4e1

+400.00e-2

4000E-3
```

EJEMPLO Algunos ejemplos de constantes de tipo real incorrectas son:

`-200` (No aparece el punto ni el exponente)

`-20,0` (No puede aparecer la coma)

`--111.` (No se puede duplicar el signo)

`-111..` (No se puede duplicar el punto)

`-111.11.` (No puede aparecer más de un punto)

`+22e` (Después del carácter (`e`) o (`E`) se debe escribir el exponente)

`+22ee6` (No se puede duplicar el carácter (`e`) o (`E`))

`+22e 6` (No se puede escribir el carácter espacio en blanco)

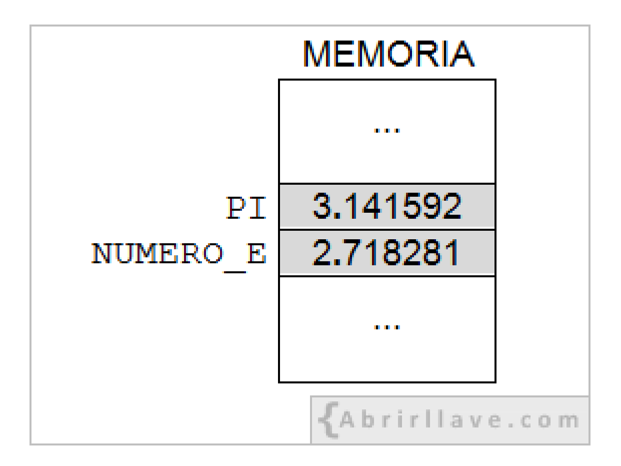
`38E-2.2` (El exponente debe ser una cantidad entera)

EJEMPLO El número π (pi) y el número *e*, son dos ejemplos de valores reales –datos– frecuentemente declarados como constantes en los programas que hacen uso de ellos.

```
PI = 3.141592

NUMERO_E = 2.718281
```

El número *e* es el límite de la expresión $(1+1/n)^n$ cuando n tiende a infinito.



Han aparecido tres nuevos símbolos reservados.

Símbolos reservados	
Símbolo	***Descripción***
`.`	Separador de la parte entera y decimal de un número real.
`e`	Separador de un número real y su exponente.
`E`	Separador de un número real y su exponente.

3.3.3. Constantes de tipo lógico

Una **constante de tipo lógico** es aquella que representa a un valor –dato– perteneciente al conjunto:

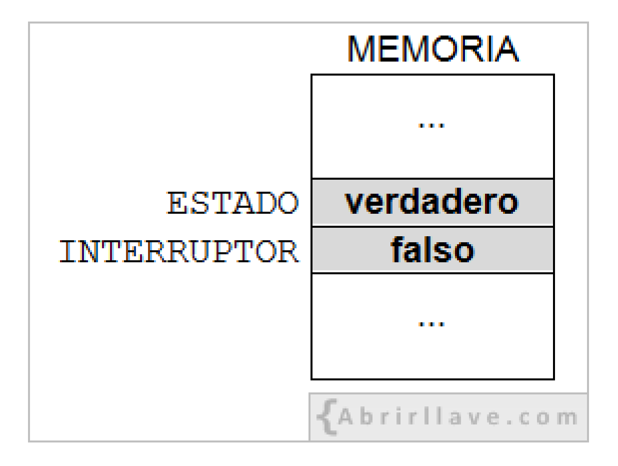
{ **`verdadero`**, **`falso`** }

`verdadero` y **`falso`** son palabras reservadas –identificadores– que, en sí mismas, representan a constantes de tipo lógico. En consecuencia, aunque se pueden definir constantes de tipo lógico, no suele ser habitual declarar constantes de este tipo de dato:

EJEMPLO

```
ESTADO = verdadero

INTERRUPTOR = falso
```



De cualquier modo, como ya vimos en el capítulo 2 "Tipos de datos", en lenguaje C no existe el tipo de dato lógico.

3.3.4. Constantes de tipo carácter

Una **constante de tipo carácter** es aquella que representa a un valor –dato– perteneciente al conjunto de caracteres que puede representar el ordenador.

EJEMPLO Las siguientes constantes de tipo carácter están expresadas por su valor:

```
'a'

'T'

'5'

'+'

'.'
```

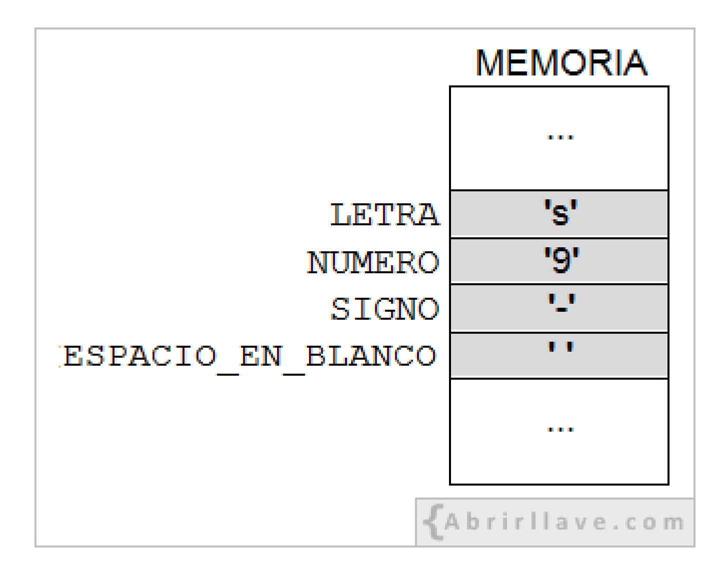
EJEMPLO Algunos ejemplos de declaración de constantes de tipo carácter son:

```
LETRA = 's'

NUMERO = '9'

SIGNO = '-'

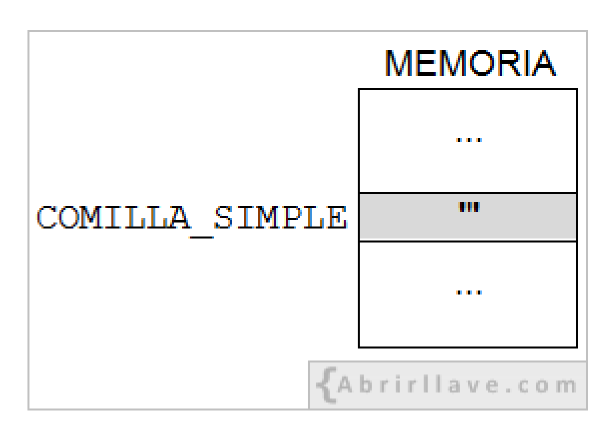
ESPACIO_EN_BLANCO = ' '
```



EJEMPLO En lenguaje C, y en nuestro pseudocódigo CEE, para representar el carácter comilla simple (`'`), se debe anteponer el carácter barra invertida (`\`).



```
COMILLA_SIMPLE = '\''
```



Han aparecido dos nuevos símbolos reservados.

Símbolos reservados	
Símbolo	***Descripción***
'	Se escribe delante y detrás de un valor de tipo carácter.
\	Se escribe delante del carácter comilla simple (') para representarlo como un carácter.

3.3.5. Constantes de tipo cadena

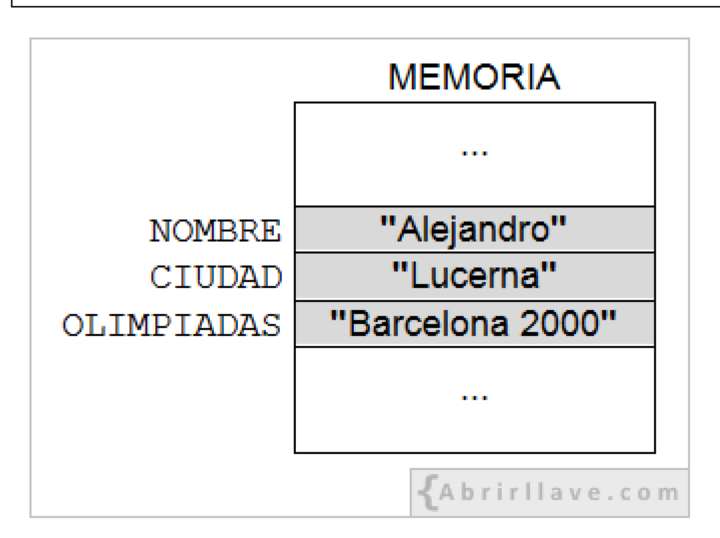
Una **constante de tipo cadena** es aquella que representa a un valor –dato– de tipo cadena, es decir, representa a una secuencia de caracteres.

EJEMPLO Las siguientes constantes de tipo cadena están expresadas por su valor:

```
"Alejandro"
"Lucerna"
"Barcelona 2000"
```

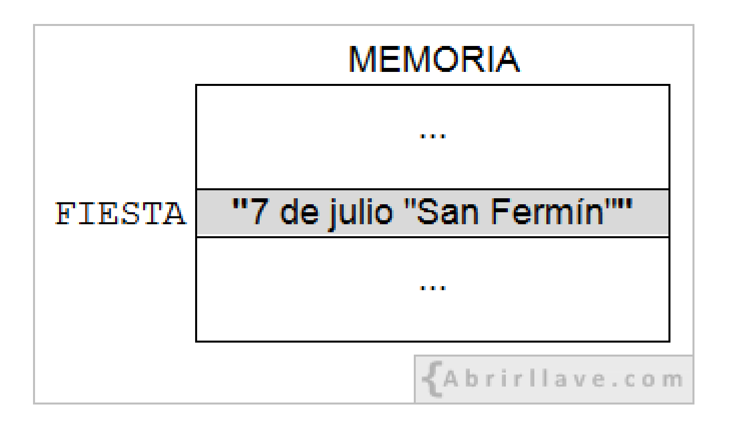
EJEMPLO Algunos ejemplos de declaración de constantes de tipo cadena son:

```
NOMBRE = "Alejandro"
CIUDAD = "Lucerna"
OLIMPIADAS = "Barcelona 2000"
```



EJEMPLO En lenguaje C, y en nuestro pseudocódigo CEE, para representar el carácter comilla doble (") dentro de una cadena, se debe anteponer el carácter barra invertida (\).

```
FIESTA = "7 de julio \"San Fermín\""
```



Ha aparecido un nuevo símbolo reservado.

"	Se escribe delante y detrás de un valor de tipo cadena.

Y el símbolo reservado barra invertida (\) ha vuelto a aparecer.

\	Se escribe delante del carácter comilla doble (") para representarlo dentro de una cadena.

Ejercicios resueltos
• Valores almacenados en memoria (de tipos definidos por el programador) • Declaraciones correctas (de tipos definidos por el programador)

Capítulo 4

Tipos de datos definidos por el programador

Además de los tipos de datos simples predefinidos –vistos en el capítulo 2 "Tipos de datos"– que proporcionan los lenguajes de programación, el programador tiene la posibilidad de definir –declarar– sus propios tipos de datos. Los tipos de datos simples que puede definir el programador son:

- Enumerados.
- Subrangos.

4.1. Datos de tipos enumerados

Un **dato de un tipo enumerado** es aquel que puede tomar por valor uno de los pertenecientes a una lista ordenada de valores definida por el programador.

EJEMPLO El color de un semáforo puede ser uno de los siguientes:

{ rojo, verde, amarillo }

EJEMPLO Otro ejemplo de dato enumerado puede ser la dirección en la que se mueve un coche. Los valores son:

{ norte, sur, este, oeste }

4.1.1. Declaración de tipos enumerados

En nuestro pseudocódigo CEE, para declarar un tipo de dato enumerado, vamos a utilizar la sintaxis:

```
enumerado <nombre_del_tipo> { <constante_1> [ = <valor_1> ],
                              <constante_2> [ = <valor_2> ],
                                 ...,
                              <constante_n> [ = <valor_n> ] }
```

Como se puede apreciar, los valores de la lista se representan por medio de identificadores de constantes.

EJEMPLO Para declarar el tipo enumerado **direcciones**, se debe escribir:

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }
```

La lista de constantes está ordenada, y cada una de ellas representa a un valor entero empezando por el **0**, e incrementándose de uno en uno. De manera que, las constantes declaradas representan a los valores enteros {**0**, **1**, **2**, **3**}.

NORTE representa al valor **0**

SUR representa al valor **1**

ESTE representa al valor **2**

OESTE representa al valor **3**

Pero, dichos valores pueden ser diferentes si así se indica en la declaración.

EJEMPLO Se puede escribir:

```
enumerado direcciones { NORTE = -2, SUR, ESTE, OESTE }
```

En este caso, las constantes declaradas representan a los valores {**-2**, **-1**, **0**, **1**}, ya que, a partir de la asignación **NORTE = -2**, las demás constantes de la lista toman valores incrementándose de uno en uno.

NORTE representa al valor **-2**

SUR representa al valor **-1**

ESTE representa al valor **0**

OESTE representa al valor **1**

EJEMPLO También se puede escribir, por ejemplo:

```
enumerado direcciones { NORTE, SUR, ESTE = 4, OESTE }
```

Ahora, las constantes declaradas representan a los valores {**0**, **1**, **4**, **5**}.

NORTE representa al valor **0**

SUR representa al valor **1**

ESTE representa al valor **4**

OESTE representa al valor **5**

NORTE es la primera constante de la lista, en consecuencia, representa al valor **0**. Después, **SUR** representa al valor **1**. Pero, a **ESTE**, que debería representar al valor **2**, se le ha asignado el valor **4** en la declaración, de manera que, **OESTE** representa al valor **5**. Si después hubiese otra constante en la lista, representaría al valor **6**, y así sucesivamente.

Han aparecido dos nuevos símbolos reservados.

Símbolos reservados	
Símbolo	***Descripción***
{	Se escribe delante de la lista de identificadores de constantes de un tipo de dato enumerado.
}	Se escribe detrás de la lista de identificadores de constantes de un tipo de dato enumerado.

Y el símbolo reservado coma (**,**) ha vuelto a aparecer.

,	Separadora de los identificadores de las constantes de la lista asignada a un tipo de dato enumerado.

4.1.2. Variables de tipos enumerados

Una **variable de un tipo enumerado** representa a un espacio de memoria en donde se puede almacenar un dato de un tipo enumerado.

EJEMPLO Dadas las declaraciones:

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }

direcciones direccion_de_un_coche
```

direccion_de_un_coche es una variable del tipo enumerado **direcciones**. Por tanto, en el espacio de memoria representado por la variable se podrá almacenar uno de los valores {**0**, **1**, **2**, **3**}.

EJEMPLO Las declaraciones del ejemplo anterior se pueden combinar de la forma siguiente:

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }
   direccion_de_un_coche
```

EJEMPLO También se pueden combinar prescindiendo del nombre –identificador– del tipo de dato enumerado.

```
enumerado { NORTE, SUR, ESTE, OESTE } direccion_de_un_coche
```

Varias variables del mismo tipo de dato enumerado se pueden declarar de diferentes formas. A continuación, se muestran algunos ejemplos.

EJEMPLO

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }

direcciones direccion_de_un_coche
direcciones direccion_de_un_avion = SUR
direcciones direccion_de_un_camion
```

EJEMPLO

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }

direcciones direccion_de_un_coche,
            direccion_de_un_avion = SUR,
            direccion_de_un_camion
```

EJEMPLO

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }
   direccion_de_un_coche, direccion_de_un_avion = SUR,
   direccion_de_un_camion
```

EJEMPLO

```
enumerado { NORTE, SUR, ESTE, OESTE }
   direccion_de_un_coche, direccion_de_un_avion = SUR,
   direccion_de_un_camion
```

EJEMPLO

```
enumerado direcciones { NORTE, SUR, ESTE, OESTE }
   direccion_de_un_coche

direcciones direccion_de_un_avion = SUR,
   direccion_de_un_camion
```

4.2. Datos de tipos subrangos

En lenguaje C no existen datos de tipos subrangos, ya que, el programador no puede definir tipos de datos subrango en este lenguaje. No obstante, otros lenguajes de programación sí permiten definirlos.

Un **dato de un tipo subrango** es aquel que puede tomar por valor uno de los pertenecientes a un subrango definido por el programador.

Matemáticamente, un rango es el conjunto de valores comprendidos entre un valor mínimo y un valor máximo, ambos inclusive. Por ejemplo, suponiendo que el ordenador –utilizando dieciséis bits– puede representar el siguiente conjunto de valores enteros:

{ -32768, -32767, ..., -1, 0, 1, ..., 32766, 32767 }

Los valores mínimo y máximo de ese rango son los números -32768 y 32767, respectivamente. Pues bien, un subrango es un subconjunto de valores de un rango.

EJEMPLO Del rango { -32768, ..., 0, ..., 32767 } posibles subrangos son:

{ 1, 2, 3, 4, 5, 6 }

{ 0, 1, 2, ..., 8, 9, 10 }

{ -10, -9, -8, -7 }

{ -3, -2, -1, ..., 5, 6, 7 }

{ 1240, 1241, 1243, ..., 2999, 3000, 3001 }

EJEMPLO Sin embargo, al rango { -32768, ..., 0, ..., 32767 } no pertenecen los siguientes subrangos:

{ 0, 1, 2, ... 34998, 34999, 35000 }

{ -50000, -49999, -49998, ..., 49998, 49999, 50000 }

Como se puede observar, el conjunto de valores de un subrango también está comprendido entre un valor mínimo y un valor máximo. Además, en el subrango tienen que estar todos los elementos que están en el rango entre ambos valores.

EJEMPLO Así pues, del rango { -32768, ..., 0, ..., 32767 }, los siguientes conjuntos no pueden ser considerados como subrangos:

{ 2, 4, 6, 8, 10 }

{ 1, 2, 4, 8, 16, 32 }

Todos los datos de tipos subrangos son ordinales, es decir, solamente pueden tomar por valor elementos de subrangos finitos.

4.2.1. Declaración de tipos subrangos

En pseudocódigo, para declarar un tipo de dato subrango, se puede utilizar la sintaxis:

```
<nombre_del_tipo> = <valor_mínimo>..<valor_máximo>
```

O también:

```
subrango <nombre_del_tipo> = <valor_mínimo>..<valor_máximo>
```

EJEMPLO Suponiendo que la calificación de una asignatura sea dada con un valor perteneciente al subrango { 0, 1, 2, ..., 8, 9, 10 } del rango { -32768, ..., 0, ..., 32767 }, para declarar el tipo subrango calificaciones, se debe escribir:

```
subrango calificaciones = 0..10
```

Ha aparecido un nuevo símbolo reservado.

`..`	Separador del valor mínimo y máximo de un subrango.

Y el símbolo reservado igual (**=**) ha vuelto a aparecer.

=	Separador del identificador de un tipo de dato subrango y del subrango asignado al mismo.

4.2.2. Variables de tipos subrangos

Una **variable de un tipo subrango** representa a un espacio de memoria en donde se puede almacenar un dato de un tipo subrango.

EJEMPLO Dadas las declaraciones:

```
subrango calificaciones = 0..10

calificaciones matematicas
```

`matematicas` es una variable del tipo subrango **`calificaciones`**. En consecuencia, en el espacio de memoria representado por la variable se podrá almacenar uno de los valores del conjunto { 0, 1, 2, ..., 8, 9, 10 }.

EJEMPLO Las declaraciones del ejemplo anterior se pueden combinar de la forma siguiente:

```
subrango calificaciones = 0..10 matematicas
```

EJEMPLO También, se pueden combinar prescindiendo del nombre –identificador– del tipo de dato subrango.

```
subrango 0..10 matematicas
```

Varias variables del mismo tipo de dato subrango se pueden declarar de diferentes formas. A continuación, se muestran algunos ejemplos.

EJEMPLO

```
subrango calificaciones = 0..10

calificaciones matematicas
calificaciones fisica
calificaciones quimica
```

EJEMPLO

```
subrango calificaciones = 0..10

calificaciones matematicas, fisica, quimica
```

EJEMPLO

```
subrango calificaciones = 0..10 matematicas, fisica, quimica
```

EJEMPLO

```
subrango 0..10 matematicas, fisica, quimica
```

EJEMPLO

```
subrango calificaciones = 0..10 matematicas

calificaciones fisica, quimica
```

Ejercicios resueltos

- Valores almacenados en memoria
- Declaraciones correctas de tipos enumerados y subrangos
- Test de autoevaluación

Capítulo 5

Operadores y expresiones

En un programa, el tipo de un dato determina las operaciones que se pueden realizar con él. Por ejemplo, con los datos de tipo entero se pueden realizar operaciones aritméticas, tales como la suma, la resta o la multiplicación.

EJEMPLO Algunos ejemplos son:

`111` **+** `6` (operación *suma*)

`19` **–** `72` (operación *resta*)

`24` ***** `3` (operación *multiplicación*)

Todas las operaciones del ejemplo constan de dos operandos –constantes enteras– y un operador. La mayoría de las veces es así, pero, también es posible realizar operaciones con distinto número de operadores y/u operandos.

EJEMPLO

`111` **+** `6` **–** `8` (tres operandos y dos operadores)

–((**+**`19` **)** **+** `72` **)** (dos operandos y tres operadores)

–(**–**`72` **)** (un operando y dos operadores)

–(**+**`43` **)** (un operando y dos operadores)

En las operaciones del ejemplo se puede observar que los caracteres *más* (**+**) y *menos* (**–**) tienen dos usos:

1. Operadores suma y resta.
2. Signos de un número (también son operadores).

Los operadores de signo más (**+**) y menos (**–**) son **operadores monarios**, también llamados **unarios**, ya que, actúan, solamente, sobre un operando.

Los caracteres *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**" se utilizan para establecer la prioridad de los operadores, es decir, para establecer el orden en el que los operadores actúan sobre los operandos.

EJEMPLO Obsérvese la diferencia entre las siguientes operaciones:

`-19 + 72` (dos operandos y dos operadores)

`-( 19 + 72 )` (dos operandos y dos operadores)

Los resultados de evaluarlas son, respectivamente:

53 (primero actúa el operador signo (**-**) y después el operador suma (**+**))

-91 (primero actúa el operador suma (**+**) y después el operador signo (**-**))

EJEMPLO Percátese también de las diferencias entre las siguientes operaciones:

```
( ( 3 * 7 ) + 2 ) - 1
( 3 * ( 7 + 2 ) ) - 1
3 * ( ( 7 + 2 ) - 1 )
( 3 * 7 ) + ( 2 - 1 )
```

Al evaluarlas se obtienen los valores:

22 (actúan en orden los operadores: *multiplicación* (*****), *suma* (**+**) y *resta* (**-**))

26 (actúan en orden los operadores: *suma* (**+**), *multiplicación* (*****) y *resta* (**-**))

24 (actúan en orden los operadores: *suma* (**+**), *resta* (**-**) y *multiplicación* (*****))

22 (actúan en orden los operadores: *multiplicación* (*****), *resta* (**-**) y *suma* (**+**))

Un **operador** indica el tipo de operación a realizar sobre los operandos –datos– que actúa. Los operandos pueden ser:

- Constantes, expresadas por su valor o con su nombre (identificador).
- Variables.
- Llamadas a funciones.
- Elementos de formaciones (arrays).

En este capítulo se van a tratar operaciones en donde únicamente aparecen constantes y variables. Las funciones y las formaciones se estudiarán más adelante.

Cuando se combinan uno o más operadores con uno o más operandos se obtiene una expresión. De modo que, una **expresión** es una secuencia de operandos y operadores escrita bajo unas reglas de sintaxis. Por ejemplo, dadas las siguientes declaraciones de constantes y variables en pseudocódigo:

```
PI = 3.141592

entero numero = 2

real radio_circulo = 3.2
```

EJEMPLO Algunos ejemplos de expresiones son:

```
2 * PI * radio_circulo

( PI * PI )

numero * 5
```

De sus evaluaciones se obtienen los *valores*:

20.106189 (valor real) (`2 * 3.141592 * 3.2`)

9.869600 (valor real) (`3.141592 * 3.141592`)

10 (valor entero) (`2 * 5`)

Un operador siempre forma parte de una expresión, en la cual, el operador siempre actúa sobre al menos un operando. Por el contrario, un operando sí puede aparecer solo en una expresión.

EJEMPLO Las siguientes expresiones están constituidas por un solo operando, es decir, no están bajo la influencia de ningún operador:

```
PI

numero

5
```

Los resultados de evaluarlas son:

3.141592 (valor real)

2 (valor entero)

5 (valor entero)

De la evaluación de una expresión siempre se obtiene un valor. Dicho valor puede ser de tipo entero, real, lógico, carácter o cadena. Por consiguiente, una expresión puede ser:

- Aritmética (devuelve un número entero o real).
- Lógica (devuelve un valor lógico: **`verdadero`** o **`falso`**).
- De carácter (devuelve un carácter representable por el ordenador).
- De cadena (devuelve una cadena).

Dependiendo del tipo de expresión, pueden participar unos operadores u otros.

5.1. Expresiones aritméticas

De la evaluación de una **expresión aritmética** siempre se obtiene un valor de tipo entero o real. En las expresiones aritméticas se pueden utilizar los siguientes **operadores aritméticos**:

Operadores aritméticos en pseudocódigo	
Operador	***Descripción***
+	Suma
-	Resta
*****	Multiplicación
******	Potencia (también se utiliza la *flecha arriba* **↑** o el *acento circunflejo* **^**)
/	División real
div	División entera (también se utiliza el carácter *barra invertida* ****)
mod	Módulo (resto de la división entera)
+	Signo más
-	Signo menos

EJEMPLO El **operador suma** (**+**) realiza la suma de dos operandos numéricos.

```
5 + 2

3.1 + 2.5
```

De la evaluación de las expresiones anteriores se obtienen los valores:

7 (valor entero)

5.6 (valor real)

EJEMPLO El **operador resta** (**-**) realiza la resta entre dos operandos numéricos.

```
5 - 2

3.1 - 2.5
```

Obteniéndose los valores:

3 (valor entero)

0.6 (valor real)

EJEMPLO El **operador multiplicación** (*****) realiza la multiplicación de dos operandos numéricos.

```
5 * 2

3.1 * 2.5
```

Ahora los resultados son:

10 (valor entero)

7.75 (valor real)

EJEMPLO El **operador potencia** (**`**`**) eleva el operando de la izquierda (número base) al operando de la derecha (potencia o exponente).

```
5 ** 2

3.1 ** 2.5
```

De estas expresiones, se obtienen los valores:

25 (valor entero)

16.920151 (valor real)

EJEMPLO El **operador división real** (**`/`**) realiza la división real entre dos operandos numéricos.

```
5 / 2

3.1 / 2.5
```

Sus resultados son:

2.5 (valor real)

1.24 (valor real)

EJEMPLO El **operador división entera** (**`div`**) realiza la división entera entre dos operandos numéricos enteros.

```
5 div 2

3.1 div 2.5
```

El operador división entera (**`div`**) no puede operar con operandos numéricos reales. Por tanto, al evaluar las expresiones de este ejemplo se obtienen los valores:

2 (entero)

ERROR (no se puede evaluar; ambos operandos deben ser valores enteros)

EJEMPLO El **operador módulo** (**mod**) realiza la división entera entre dos operandos numéricos enteros, devolviendo el resto de la misma.

```
5 mod 2

3.1 mod 2.5
```

Al igual que el operador división entera (**div**), el operador módulo (**mod**) tampoco puede operar con operandos numéricos reales. De modo que, en este caso, los resultados son:

1 (valor entero)

ERROR (no se puede evaluar; ambos operandos deben ser valores enteros)

EJEMPLO El **operador signo menos** (**-**) cambia el signo de un operando numérico. Así, de las expresiones:

```
-11

-( 3.1 )

-( -2.5 )
```

Se obtienen los valores:

-11 (valor entero)

-3.1 (valor real)

2.5 (valor real)

EJEMPLO El **operador signo más** (**+**) mantiene el signo de un operando numérico. Así, de las expresiones:

```
+11

+( 3.1 )

+( -2.5 )
```

Los valores que se obtienen son:

11 (valor entero)

3.1 (valor real)

-2.5 (valor real)

EJEMPLO En una expresión aritmética pueden aparecer operandos numéricos enteros y reales al mismo tiempo.

```
11 + 3.1

2.5 - 3

-3.1 * 10

11 ** 2.5

+3.1 / 2
```

De estas expresiones se obtienen los valores:

14.1 (valor real)

-0.5 (valor real)

-31.0 (valor real)

401.311600 (valor real)

1.55 (valor real)

Como se puede apreciar, al combinar operandos numéricos reales con operandos numéricos enteros en una expresión aritmética, el resultado de su evaluación siempre es un valor numérico real, excepto con el operador módulo (**mod**) o con el operador división entera (**div**), en cuyos casos se obtendrá ERROR.

5.1.1. Prioridad de los operadores aritméticos

La prioridad de los operadores puede variar de unos lenguajes a otros, pero, en pseudocódigo, en este tutorial, vamos a establecer una prioridad de operadores muy similar a la que se aplica en lenguaje C. La prioridad no puede ser exactamente la misma, ya que, en C existen algunos operadores que no existen en pseudocódigo, y al revés.

EJEMPLO En una expresión aritmética puede aparecer más de un operador aritmético.

`11 + 3 div 3` (dos operadores)

`-3 * 6 mod 4` (tres operadores)

`-3.1 + 5 * 0.5` (tres operadores)

`3 ** 3 - 1` (dos operadores)

`+3 * -8` (tres operadores)

Para poder evaluar correctamente las expresiones aritméticas del ejemplo, es necesario seguir un criterio de prioridad de operadores. En nuestro pseudocódigo CEE, la prioridad de los operadores aritméticos es:

Prioridad de los operadores aritméticos (de mayor a menor) en pseudocódigo	
Operadores	***Descripción***
+ -	Signos más y menos
**	Potencia
*** / div mod**	Multiplicación, división real, división entera y módulo
+ -	Suma y resta

A excepción de los operadores de signo, que se evalúan de derecha a izquierda en una expresión, todos los demás operadores aritméticos con la misma prioridad, por ejemplo, el operador multiplicación (*****) y el operador módulo (**mod**), se evalúan de izquierda a derecha. En consecuencia, los valores que proporcionan las expresiones del ejemplo anterior son:

12 (actúan en orden los operadores: (**div**) y suma (**+**))

-2 (actúan en orden los operadores: signo menos (**-**), (*****) y (**mod**))

-0.6 (actúan en orden los operadores: signo menos (**-**), (*****) y suma (**+**))

26 (actúan en orden los operadores: (******) y resta (**-**))

-24 (actúan en orden los operadores: signo menos (**-**), signo más (**+**) y (*****))

Para modificar la prioridad de los operadores en las expresiones, se debe hacer uso de los caracteres *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**".

EJEMPLO Para cambiar la prioridad de los operadores de las expresiones del ejemplo anterior, se puede escribir:

```
( 11 + 3 ) div 3

-3 * ( 6 mod 4 )

( -3.1 + 5 ) * 0.5

3 ** ( 3 - 1 )

( +3 ) * -8
```

De la evaluación de estas expresiones se obtienen los valores:

4 (actúan en orden los operadores: suma (**+**) y (**div**))

-6 (actúan en orden los operadores: (**mod**), signo menos (**-**) y (*****))

0.95 (actúan en orden los operadores: signo menos (**-**), suma (**+**) y (*****))

9 (actúan en orden los operadores: resta (**-**) y (******))

-24 (actúan en orden los operadores: signo más (**+**), signo menos (**−**) y (*****))

En las expresiones aritméticas hay que tener la precaución de no dividir entre cero (0).

EJEMPLO Por tanto, las siguientes expresiones son incorrectas:

```
11 / 0

5 div 0

-3 mod 0
```

De la evaluación de cada una de estas expresiones se obtiene:

ERROR (no se puede evaluar; no se puede dividir entre cero)

5.2. Expresiones lógicas

De la evaluación de una **expresión lógica** siempre se obtiene un valor de tipo lógico (**verdadero** o **falso**). En las expresiones lógicas se pueden utilizar dos tipos de operadores:

- Relacionales.
- Lógicos.

Un **operador relacional** se utiliza para comparar los valores de dos expresiones. Estas deben ser del mismo tipo (aritméticas, lógicas, de carácter o de cadena).

EJEMPLO Algunos ejemplos son:

`22 > 13` (comparación de dos expresiones aritméticas)

`22.5 < 3.44` (comparación de dos expresiones aritméticas)

`verdadero = falso` (comparación de dos expresiones lógicas)

`'c' > 'f'` (comparación de dos expresiones de carácter)

`"coche" = "Coche"` (comparación de dos expresiones de cadena)

Proporcionan los valores:

verdadero (22 es mayor que 13)

falso (22.5 no es menor que 3.44)

falso (**verdadero** no es igual que **falso**)

falso ('c' no es mayor que 'f')

falso ("coche" no es igual que "Coche")

Las comparaciones entre los valores de tipo numérico son obvias. En cuanto a los valores de tipo lógico (**verdadero** y **falso**) se considera que **falso** es menor que **verdadero**. En lo que respecta a los valores de tipo carácter, su orden viene dado por el ASCII extendido utilizado por el ordenador para representarlos. Y en el caso de los valores de tipo cadena, también se tiene en cuenta dicho código.

Los operadores relacionales son:

Operadores relacionales en pseudocódigo	
Operador	***Descripción***
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
=	Igual que
<>	Distinto que

Para escribir una **expresión relacional** (lógica) se utiliza la sintaxis:

```
<expresión_1> <operador_de_relación> <expresión_2>
```

Siendo **<expresión_1>** y **<expresión_2>** del mismo tipo (aritmética, lógica, de carácter o de cadena).

EJEMPLO La comparación entre valores de tipo cadena se realiza carácter a carácter.

"a" **=** "b" (se compara "a" con "b")

"bc" **<>** "kd" (se compara "b" con "k" y "c" con "d")

"126" **<** "9" (se compara "1" con "9")

"ab" **<=** "ab" (se compara "a" con "a" y "b" con "b")

"abb" **>=** "abc" (se compara "a" con "a", "b" con "b" y "b" con "c")

De estas expresiones se obtienen los valores:

falso ("a" no es igual que "b")

verdadero ("bc" es distinto que "kd")

verdadero ("1" es menor que "9")

verdadero ("ab" es menor o igual que "ab")

falso ("abb" no es mayor o igual que "abc")

Un **operador lógico** actúa, exclusivamente, sobre valores de expresiones lógicas. Los operadores lógicos son:

Operadores lógicos en pseudocódigo	
Operador	***Descripción***
y	Conjunción
o	Disyunción
no	Negación

El operador conjunción (**y**) y el operador disyunción (**o**) siempre actúan sobre dos operandos, mientras que, el operador negación (**no**) solamente actúa sobre un operando, o dicho de otra forma, es un operador monario.

El modo en que actúan los operadores lógicos se resume en las llamadas tablas de verdad, definidas por el matemático George Boole.

La tabla de verdad del **operador conjunción** (**y**) es:

Tabla de verdad del operador conjunción (y)		
<expresión_1>	**<expresión_2>**	**<expresión_1> y <expresión_2>**
verdadero	**verdadero**	**verdadero**
verdadero	**falso**	**falso**
falso	**verdadero**	**falso**
falso	**falso**	**falso**

Se supone que **<expresión_1>** y **<expresión_2>** son expresiones lógicas. De la tabla de verdad se deduce que **<expresión_1> y <expresión_2>** se evalúa a **verdadero** solamente en el caso de que tanto **<expresión_1>** como **<expresión_2>** se evalúen también como verdaderas, en cualquier otro caso el resultado será **falso**. Dicho de otro modo, si al menos una de las dos expresiones es falsa, el resultado será **falso**.

EJEMPLO Algunos ejemplos son:

```
9 > 3 y 8 > 6

9 > 3 y 8 > 9

9 = 3 y 8 >= 6

9 = 3 y 8 >= 9
```

Las expresiones anteriores se evalúan a:

verdadero (9 **>** 3 es **verdadero** y 8 **>** 6 es **verdadero**)

falso (9 **>** 3 es **verdadero** y 8 **>** 9 es **falso**)

falso (9 **=** 3 es **falso** y 8 **>=** 6 es **verdadero**)

falso (9 **=** 3 es **falso** y 8 **>=** 9 es **falso**)

La tabla de verdad del **operador disyunción** (**o**) es:

Tabla de verdad del operador disyunción (o)		
<expresión_1>	**<expresión_2>**	**<expresión_1> o <expresión_2>**
verdadero	**verdadero**	**verdadero**
verdadero	**falso**	**verdadero**
falso	**verdadero**	**verdadero**
falso	**falso**	**falso**

Se supone que **<expresión_1>** y **<expresión_2>** son expresiones lógicas. De la tabla de verdad se deduce que **<expresión_1> o <expresión_2>** se evalúa a **falso** solamente en el caso de que tanto **<expresión_1>** como **<expresión_2>** se evalúen también como falsas, en cualquier otro caso el resultado será **verdadero**. Dicho de otro modo, si al menos una de las dos expresiones es verdadera, el resultado será **verdadero**.

EJEMPLO Algunos ejemplos son:

```
9 > 3 o 8 > 6

9 > 3 o 8 > 9

9 = 3 o 8 >= 6

9 = 3 o 8 >= 9
```

Las expresiones anteriores se evalúan:

verdadero (9 **>** 3 es **verdadero** y 8 **>** 6 es **verdadero**)

verdadero (9 **>** 3 es **verdadero** y 8 **>** 9 es **falso**)

verdadero (9 **=** 3 es **falso** y 8 **>=** 6 es **verdadero**)

falso (9 **=** 3 es **falso** y 8 **>=** 9 es **falso**)

La tabla de verdad del **operador negación** (**no**) es:

Tabla de verdad del operador negación (no)	
<expresión>	**no <expresión>**
verdadero	**falso**
falso	**verdadero**

Se supone que **<expresión>** es una expresión lógica, al evaluarla se obtiene el valor **verdadero** o el valor **falso**. Y como se puede apreciar en la tabla, el valor de **no <expresión>** es el contrario al valor obtenido de **<expresión>**.

EJEMPLO De las expresiones:

```
no ( 9 > 3 )

no ( 8 > 9 )
```

Los resultados de evaluarlas son:

falso (9 **>** 3 es **verdadero**)

verdadero (8 **>** 9 es **falso**)

5.2.1. Prioridad de los operadores relacionales y lógicos

En una expresión lógica puede aparecer uno o más operadores relacionales y/o lógicos.

EJEMPLO Algunos ejemplos son:

```
3 > 1 o 4 < 1 y 4 <= 2

no falso y falso

verdadero >= verdadero = falso

falso = verdadero <= verdadero
```

Para poder evaluar correctamente las expresiones lógicas del ejemplo, es necesario seguir un criterio de prioridad de operadores. En nuestro pseudocódigo CEE, la prioridad entre los operadores relacionales y lógicos es:

Prioridad de los operadores relacionales y lógicos (de mayor a menor) en pseudocódigo	
Operadores	***Descripción***
no	Negación
< <= > >=	Menor que, menor o igual que, mayor que, mayor o igual que
= <>	Igual que, distinto que
y	Conjunción
o	Disyunción

A excepción del operador negación (**no**), que se evalúa de derecha a izquierda en una expresión, todos los demás operadores con la misma prioridad, por ejemplo, el operador menor que (**<**) y el operador mayor que (**>**), se evalúan de izquierda a derecha. Así que, los valores que proporcionan las expresiones del ejemplo anterior son:

verdadero (actúan en orden los operadores: (**>**), (**<**), (**<=**), (**y**) y (**o**))

falso (actúan en orden los operadores: (**no**) e (**y**))

falso (actúan en orden los operadores: (**>=**) y (**=**))

falso (actúan en orden los operadores: (**<=**) y (**=**))

De nuevo, podemos hacer uso de los caracteres *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**" para modificar la prioridad de los operadores.

EJEMPLO Para cambiar la prioridad de los operadores de las expresiones del ejemplo anterior, se puede escribir:

```
( 3 > 1 o 4 < 1 ) y 4 <= 2

no ( falso y falso )

verdadero >= ( verdadero = falso )

( falso = verdadero ) <= verdadero
```

De la evaluación de estas expresiones se obtienen los valores:

falso (actúan en orden los operadores: (**>**), (**<**), (**o**), (**<=**) e (**y**))

verdadero (actúan en orden los operadores: (**y**) y (**no**))

verdadero (actúan en orden los operadores: (**=**) y (**>=**))

verdadero (actúan en orden los operadores: (**=**) y (**<=**))

5.3. Expresiones de carácter

Aunque no existe ningún operador de caracteres, sí que existen expresiones de carácter. De la evaluación de una **expresión de carácter** siempre se obtiene un valor de tipo carácter.

EJEMPLO Dadas las siguientes declaraciones de constantes y variables en pseudocódigo:

```
CONSONANTE = 'S'

caracter letra = 'X'

caracter opcion = '3'
```

Algunas expresiones de carácter son:

```
opcion

letra

CONSONANTE

'a'
```

Los resultados de evaluarlas son:

```
'3'

'X'

'S'

'a'
```

5.4. Expresiones de cadena

De la evaluación de una **expresión de cadena** siempre se obtiene un valor de tipo cadena. Solamente existe un operador de cadena:

Operador de cadena en pseudocódigo	
Operador	***Descripción***
+	Concatenación

El **operador concatenación** (**+**) realiza la concatenación de dos operandos de tipo cadena, es decir, los encadena.

EJEMPLO Dadas las siguientes declaraciones de constantes y variables en pseudocódigo:

```
OLIMPIADA = "Atenas 2004"

PUNTO = "."

cadena nombre = "Pedro", apellido = "Cosín", rio = "Tajo"
```

Algunas expresiones de cadena son:

```
OLIMPIADA + PUNTO

nombre + " " + apellido

"Buenos días" + PUNTO

rio

nombre + " fue a las Olimpiadas de " + OLIMPIADA + PUNTO
```

Los resultados de evaluarlas son:

```
"Atenas 2004."

"Pedro Cosín"

"Buenos días."

"Tajo"

"Pedro fue a las Olimpiadas de Atenas 2004."
```

5.5. Prioridad de los operadores aritméticos, relacionales, lógicos y de cadena

En una expresión puede aparecer uno o más operadores aritméticos, relacionales, lógicos y/o de cadena.

EJEMPLO Algunos ejemplos son:

```
5 * 4 > 5 + 4 o falso y "ab" < "aa"

( 5 * 4 > 5 + 4 o falso ) y 'f' < 'b'

no verdadero < falso

no ( verdadero < falso )
```

Para poder evaluar correctamente las expresiones anteriores, es necesario seguir un criterio de prioridad de operadores. En nuestro pseudocódigo CEE, la prioridad entre los operadores aritméticos, relacionales, lógicos y de cadena es:

Prioridad de los operadores aritméticos, relacionales, lógicos y de cadena (de mayor a menor) en pseudocódigo	
Operadores	***Descripción***
+ - no	Signo más, signo menos y negación
**	Potencia
*** / div mod**	Multiplicación, división real, división entera y módulo
+ -	Suma y resta
+	Concatenación
< <= > >=	Menor que, menor o igual que, mayor que, mayor o igual que
= <>	Igual que, distinto que
y	Conjunción
o	Disyunción

Por tanto, los valores que proporcionan las expresiones del ejemplo anterior son:

verdadero (actúan en orden los operadores: (*), suma (+), (>), (<), (y) y (o))

falso (actúan en orden los operadores: (*), suma (+), (>), (o), (<) e (y))

falso (actúan en orden los operadores: (no) y (<))

verdadero (actúan en orden los operadores: (<) y (no))

Obsérvese que, los paréntesis "()" son capaces de cambiar el orden de actuación de los operadores de cualquier expresión. Además, los paréntesis se pueden anidar, es decir, se pueden escribir unos dentro de otros, priorizándose del más interno al más externo y, después, de izquierda a derecha.

EJEMPLO De la expresión:

```
42 mod ( ( 4 - 5 ) * ( 8 + 2 ) )
```

Se obtiene el valor:

2 (actúan en orden los operadores: (-), (+), (*) y (mod))

EJEMPLO Sin embargo, de la expresión:

```
42 mod ( 4 - 5 * 8 + 2 )
```

Se obtiene el valor:

8 (actúan en orden los operadores: (*), (-), (+) y (**mod**))

Han aparecido nuevas palabras reservadas: **div**, **mod**, **no**, **o** e **y**.

También han aparecido nuevos símbolos reservados.

Símbolos reservados	
Símbolo	***Descripción***
()	Modifican la prioridad de los operadores de una expresión.
*	Operador multiplicación.
**	Operador potencia.
/	Operador división real.
<	Operador menor que.
<=	Operador menor o igual que.
>	Operador mayor que.
>=	Operador mayor o igual que.
=	Operador igual que.
<>	Operador distinto que.

Además, algunos símbolos reservados han vuelto a aparecer con significados distintos.

+	Operador suma.
+	Operador concatenación.
-	Operador resta.

Ejercicios resueltos
• Evaluación de expresiones aritméticas • Evaluación de expresiones lógicas • Evaluación de distintas expresiones

Capítulo 6

Instrucciones primitivas

En programación, las instrucciones que se utilizan para diseñar algoritmos se pueden clasificar en:

- Primitivas.
- De control.
- Llamadas a subalgoritmos (llamadas a subprogramas).

En este capítulo se van a explicar las instrucciones primitivas. El resto serán estudiadas más adelante. Existen tres tipos de instrucciones primitivas:

- Asignación.
- Salida.
- Entrada.

6.1. Instrucción de asignación

Una **instrucción de asignación** –o simplemente asignación– consiste en asignar el resultado de la evaluación de una expresión a una variable.

EJEMPLO A partir de la definición de las siguientes declaraciones de variables en pseudocódigo:

```
cadena nombre

real nota_1, nota_2, nota_3, nota_media
```

Algunas instrucciones de asignación son:

```
nota_1 ← 6.5

nota_2 ← 8.3

nota_3 ← 7.1

nota_media ← ( nota_1 + nota_2 + nota_3 ) / 3

nombre ← "Jorge"
```

En pseudocódigo, la sintaxis para escribir una asignación es:

```
<nombre_de_la_variable> ← <expresión>
```

El valor –dato– que se obtiene al evaluar la **<expresión>** es almacenado en la variable que se indique. De manera que, las variables **nota_1**, **nota_2**, **nota_3**, **nota_media** y **nombre** almacenarán los valores: 6.5, 8.3, 7.1, 7.3 y "Jorge", respectivamente.

Recuérdese que, en la declaración de una variable, también se puede asignar un valor inicial a la misma, y que, en la declaración de una constante es obligatorio asignárselo.

EJEMPLO Dadas las declaraciones:

```
PI = 3.141592

real area, longitud, radio = 5.78
```

Algunas instrucciones de asignación son:

```
area ← PI * radio ** 2

longitud ← 2 * PI * radio
```

Por consiguiente, las variables **area** y **longitud** almacenarán los valores:

57.046290 (se obtiene de `3.141592 * 5.78 ** 2`)

36.316804 (se obtiene de `2 * 3.141592 * 5.78`)

En una asignación, la variable debe ser del mismo tipo que la expresión asignada.

EJEMPLO Por tanto, partiendo de:

```
cadena telefono

entero numero
```

Las siguientes instrucciones son incorrectas:

```
telefono ← 948347788

numero ← "5"
```

Sin embargo, entre valores numéricos –enteros y reales– se puede realizar una ***conversión de tipos***.

EJEMPLO Habiendo declarado las variables:

```
real a = 6.4, b = 3.1, c, d

entero e = 5, f = 2, g, h, i
```

Después de las instrucciones:

```
c ← e / f

d ← a / f

g ← e / f

h ← a / f

i ← b / a
```

Las variables **c**, **d**, **g**, **h** e **i** contendrán, respectivamente, los valores:

2.5 (se obtiene de `5 / 2`)

3.2 (se obtiene de `6.4 / 2`)

2 (se produce un truncamiento de la parte decimal del número `2.5`)

3 (se produce un truncamiento de la parte decimal del número `3.2`)

0 (se produce un truncamiento de la parte decimal del número `0.484375`)

Una asignación permite cambiar el valor –dato– almacenado en una variable.

EJEMPLO Si se ha definido la variable:

```
entero numero = 6
```

Tras la instrucción:

```
numero ← numero * -3
```

El valor –dato– almacenado en la variable **numero** ha pasado a ser el:

-18 (se obtiene de `6 * -3`)

Como se puede observar, en esta ocasión, a la variable **numero** se le asigna el resultado de evaluar una expresión, en donde la propia variable también aparece.

Un error frecuente que suelen cometer programadores principiantes, es incluir en una expresión, una variable que no tenga ningún valor –dato– almacenado, es decir, una variable a la que previamente no se le haya asignado ningún valor.

EJEMPLO A partir de la declaración:

```
real n1, n2
```

En la siguiente instrucción:

```
n1 ← n2 * 72
```

La expresión `n2 * 72` no se puede evaluar, ya que, ¿cuál es valor de `n2`? Tiene un valor indeterminado y, en consecuencia, la instrucción se ejecutará mal.

EJEMPLO Dadas las declaraciones:

```
entero n1 = -7, n2 = 8, n3

logico negativo
```

Las siguientes asignaciones también son incorrectas:

`n1 + 1 ← n2` (ERROR de sintaxis)

`negativo ← n3 < 0` (¿cuál es el valor de `n3`?)

Ha aparecido el símbolo reservado *flecha izquierda* (←).

←	Separadora de la variable y de la expresión asignada en una instrucción de asignación.

6.2. Instrucción de salida

Una **instrucción de salida** –o simplemente **salida**– consiste en llevar hacia el exterior los valores –datos– obtenidos de la evaluación de una lista de expresiones. Normalmente, los datos son enviados a la salida estándar –la pantalla–, pero, también existen otros dispositivos de salida (la impresora, el plotter...).

En pseudocódigo, la sintaxis de una instrucción de salida es:

```
escribir( <expresión_1>, <expresión_2>, ..., <expresión_n> )
```

También se puede escribir como:

```
escribir( <lista_de_expresiones> )
```

EJEMPLO Partiendo de las variables:

```
cadena nombre = "Timoteo"

entero edad = 27, hijos = 2
```

Al escribir:

```
escribir( nombre, " tiene ", edad, " años." )

escribir( nombre, " tiene ", hijos, " hijos." )
```

Por pantalla aparecerá:

```
Timoteo tiene 27 años.

Timoteo tiene 2 hijos.
```

{Abrirllave.com

EJEMPLO Obsérvese la diferencia de escribir:

```
escribir( nombre, "tiene", edad, "años." )

escribir( "nombre tiene hijos hijos." )
```

En este caso, por pantalla se verá:

```
Timoteotiene27años.

nombre tiene hijos hijos.
```

{Abrirllave.com

De modo que, se debe ser muy cuidadoso a la hora de escribir la lista de expresiones de una salida.

EJEMPLO Si se han definido:

```
real altura, area_triangulo, base
```

Al escribir:

```
base ← 1.1

altura ← 2.2

area_triangulo ← base * altura / 2

escribir( "El área del triángulo es: ", area_triangulo )
```

Por pantalla se mostrará:

```
El área del triángulo es: 1.21
```

{Abrirllave.com

EJEMPLO El mismo resultado se puede obtener escribiendo:

```
base ← 1.1

altura ← 2.2

escribir( "El área del triángulo es: ", base * altura / 2 )
```

En este caso, se ha prescindido de la variable **area_triangulo**, reduciéndose así el número de instrucciones.

6.3. Instrucción de entrada

Una **instrucción de entrada** –o simplemente **entrada**– consiste en asignar a una o más variables, uno o más valores –datos– recibidos desde el exterior. Normalmente, los datos son recogidos desde la entrada estándar –el teclado–, pero, también existen otros dispositivos de entrada (el ratón, el escáner...).

En pseudocódigo, la sintaxis de una instrucción de entrada es:

```
leer( <nombre_de_la_variable_1>,
      <nombre_de_la_variable_2>,
       ...,
      <nombre_de_la_variable_n> )
```

También se puede escribir como:

```
leer( <lista_de_variables> )
```

EJEMPLO Partiendo de las variables:

```
cadena nombre, apellidos

entero edad
```

Para cada una de ellas se puede recoger un valor –dato– desde el teclado, escribiendo:

```
leer( nombre )

leer( apellidos )

leer( edad )
```

Otra posibilidad es:

```
leer( nombre, apellidos, edad )
```

EJEMPLO Si se han declarado:

```
cadena nombre

real numero
```

Al escribir:

```
escribir( "Introduzca su nombre: " )

leer( nombre )

escribir( "Introduzca un número real: " )

leer( numero )

escribir( nombre, ", el doble de ", numero, " es: ", numero * 2 )
```

Por pantalla se verá algo parecido a:

```
Introduzca su nombre: Juan

Introduzca un número real: 31.4

Juan, el doble de 31.4 es: 62.8
```


Juan y 31.4 son valores –datos– leídos desde el teclado, es decir, introducidos por el usuario, y almacenados en las variables **nombre** y **numero**, respectivamente.

Han aparecido dos nuevas palabras reservadas: **escribir** y **leer**.

Además, han vuelto a aparecer los símbolos reservados *abrir paréntesis* " **(**" y *cerrar paréntesis* "**)** ", con un significado distinto.

Símbolos reservados	
Símbolo	***Descripción***
( )	Albergan las expresiones de una instrucción de salida.
( )	Albergan las variables de una instrucción de entrada.

Y el símbolo *coma* (**,**).

,	Separadora de las expresiones de una instrucción de salida.
,	Separadora de las variables de una instrucción de entrada.

Ejercicios resueltos
• Valores almacenados en la memoria después de asignaciones • Salida por pantalla • Instrucciones necesarias

Capítulo 7

Estructura de un algoritmo en pseudocódigo

La estructura de un algoritmo sirve para organizar a los elementos que aparecen en él. En pseudocódigo, todos los algoritmos tienen la misma estructura, la cual viene definida por tres secciones:

- Cabecera.
- Declaraciones.
- Cuerpo.

7.1. Cabecera de un algoritmo

En la **cabecera** de un algoritmo se debe indicar el nombre –identificador– asignado al mismo. La sintaxis es:

```
algoritmo <nombre_del_algoritmo>
```

EJEMPLO Si se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado el radio (dato real) de una circunferencia.

2º) Calcule el área de la circunferencia.

3º) Muestre por pantalla el resultado (dato real).

Nota: Área de una circunferencia = π * radio^2

El algoritmo puede llamarse **`Area_de_una_circunferencia`**. Por tanto, en la cabecera se puede escribir:

```
algoritmo Area_de_una_circunferencia
```

Nota: A menos que se especifique lo contrario, en todos los problemas del libro se va a suponer que el usuario introduce por teclado correctamente los datos que se le pidan, es decir, si se le pide, por ejemplo, un dato de tipo real, no introducirá un dato de otro tipo.

7.2. Declaraciones de un algoritmo

En esta sección se declaran las constantes, los tipos de datos y las variables que se usan en el algoritmo. La sintaxis es:

```
[ constantes
     <declaraciones_de_constantes> ]

[ tipos_de_datos
     <declaraciones_de_tipos_de_datos> ]

[ variables
     <declaraciones_de_variables> ]
```

Para resolver el problema planteado en el ejemplo anterior, es necesario declarar una constante y dos variables:

```
constantes

   PI = 3.141592

variables

   real area, radio
```

En este caso, no es necesario declarar ningún tipo de dato.

7.3. Cuerpo de un algoritmo

En el **cuerpo** se escriben todas las instrucciones del algoritmo. La sintaxis es:

```
inicio

   <instrucción_1>
   <instrucción_2>
    ...
   <instrucción_n>

fin
```

`inicio` y **`fin`** son palabras reservadas que marcan el principio y final de la sección cuerpo, que es donde está el ***bloque de instrucciones principal del algoritmo***.

El cuerpo del algoritmo **Area_de_una_circunferencia** es:

```
inicio

   escribir( "Introduzca radio: " )

   leer( radio )

   area ← PI * radio ** 2

   escribir( "El área de la circunferencia es: ", area )

fin
```

Por pantalla se verá algo parecido a:

```
Introduzca radio: 3.6

El área de la circunferencia es: 40.715032
```

Abrirllave.com

EJEMPLO El mismo resultado se puede obtener escribiendo:

```
inicio

   escribir( "Introduzca radio: " )

   leer( radio )

   escribir( "El área de la circunferencia es: ",
             PI * radio ** 2 )
fin
```

Véase que la variable **area** no es necesaria. Por tanto, dicha variable no se definiría.

EJEMPLO Todavía se puede reducir más el código del algoritmo, escribiendo:

```
inicio

   escribir( "Introduzca radio: " )

   leer( radio )

   escribir( "El área de la circunferencia es: ",
             3.141592 * radio ** 2 )
fin
```

Así que, tampoco es necesario declarar la constante **PI**, reduciéndose todavía más el código. En resumen, el algoritmo completo es:

```
algoritmo Area_de_una_circunferencia

variables
   real radio

inicio
   escribir( "Introduzca radio: " )
   leer( radio )
   escribir( "El área de la circunferencia es: ",
             3.141592 * radio ** 2 )
fin
```

La sintaxis completa para escribir un algoritmo en pseudocódigo es:

```
algoritmo <nombre_del_algoritmo>

[ constantes
     <declaraciones_de_constantes> ]
[ tipos_de_datos
     <declaraciones_de_tipos_de_datos> ]
[ variables
     <declaraciones_de_variables> ]

inicio
   <bloque_de_instrucciones>
fin
```

Han aparecido seis nuevas palabras reservadas: **algoritmo**, **constantes**, **fin**, **inicio**, **tipos_de_datos** y **variables**.

Las instrucciones de los algoritmos vistos hasta ahora se ejecutan secuencialmente, una detrás de otra. Además, también el código de la sección de declaraciones se ejecuta en orden. Por ejemplo, dados los siguientes algoritmos:

EJEMPLO

```
algoritmo Ejemplo_similar_1

variables
   entero a, b = 4

inicio
   a ← b * 6
   escribir( a )
fin
```

EJEMPLO

```
algoritmo Ejemplo_similiar_2

variables
   entero a = 4, b = a * 6

inicio
   escribir( b )
fin
```

EJEMPLO

```
algoritmo Ejemplo_similar_3

variables
   entero b = a * 6, a = 4

inicio
   escribir( b )
fin
```

Tanto **`Ejemplo_similar_1`** como **`Ejemplo_similar_2`** son algoritmos que están escritos correctamente y, en ambos casos, su salida por pantalla es la misma: 24

Sin embargo, el algoritmo **`Ejemplo_similar_3`** no se puede ejecutar, debido a que, cuando se intenta evaluar la expresión `a * 6`, ¿quién es `a`? Debe definirse con anterioridad a usarse, como sí se ha hecho en el algoritmo **`Ejemplo_similar_2`**.

7.4. Comentarios en un algoritmo

En los algoritmos es conveniente escribir comentarios para explicar el diseño y/o funcionamiento del mismo. Para delimitar los comentarios se pueden utilizar distintos caracteres:

- ([) y (])
- ({) y (})
- (/*) y (*/)
- ...

En pseudocódigo, en este libro, los comentarios se van a escribir entre los símbolos reservados *barra-asterisco* (**`/*`**) y *asterisco-barra* (**`*/`**), que son los mismos que se utilizan en lenguaje C.

EJEMPLO Para comentar las secciones del algoritmo **Area_de_una_circunferencia** se puede escribir:

```
/* Cabecera */

algoritmo Area_de_una_circunferencia

/* Declaraciones */

variables
   real radio

/* Cuerpo */

inicio
   escribir( "Introduzca radio: " )
   leer( radio )
   escribir( "El área de la circunferencia es: ",
             3.141592 * radio ** 2 )
fin
```

Cuando un algoritmo se convierta –codifique– en un programa, también se podrán escribir los comentarios en el código fuente de dicho programa. Dichos comentarios no afectarán nunca a la ejecución del programa. No obstante, serán muy útiles a la hora de querer saber qué hace un algoritmo (o programa), y cómo lo hace.

Los comentarios de un algoritmo (o programa) forman parte de la documentación del mismo, pudiendo:

- Informar sobre algunos datos relevantes del algoritmo (autor, fecha de creación, fecha de última modificación, proyecto en el que se integra, versión...).
- Explicar la utilidad de uno o más tipos de datos, constantes y/o variables.
- Describir el funcionamiento general del algoritmo (o programa).
- Explicar el cometido de una o más instrucciones.
- Etc.

EJEMPLO

```
/************************************************/
/* Programa: Calcular_area_circunferencia         */
/*                                                */
/* Descripción: Recibe por teclado el radio de una */
/* circunferencia, mostrando su área por pantalla. */
/*                                                */
/* Autor: Carlos Pes                              */
/*                                                */
/* Fecha: 31/03/2005                              */
/************************************************/

/* Cabecera */

algoritmo Area_de_una_circunferencia

/* Declaraciones */

variables
   real radio

/* Cuerpo */

inicio
   escribir( "Introduzca radio: " )
   leer( radio )
   escribir( "El área de la circunferencia es: ",
             3.141592 * radio ** 2 )
fin
```

Han aparecido nuevos símbolos reservados:

Símbolos reservados	
Símbolos	***Descripción***
`/*`	Se escribe al principio de un comentario.
`*/`	Se escribe al final de un comentario.

7.5. Presentación escrita

A la hora de escribir un algoritmo, se debe intentar que su presentación escrita sea lo más legible posible.

EJEMPLO ¿Cuál es la salida por pantalla del siguiente algoritmo?

```
algoritmo Ejemplo_no_legible constantes c1 = 3, c2 =      2,
c3 = 4 variables entero  v1         =5,         v2
  =            7 inicio            v2         ←v2
+c1*c2     escribir                    (          "v2: "
,       v2       )v2              ←          v1
*           c2                                 *
c3-v2 escribir( "v2: ",   v2                        )fin
```

El algoritmo está escrito correctamente. Sin embargo, es muy difícil entender qué hace. Para ello, es mejor rescribirlo de una forma más legible.

EJEMPLO El algoritmo del ejemplo anterior se puede escribir así:

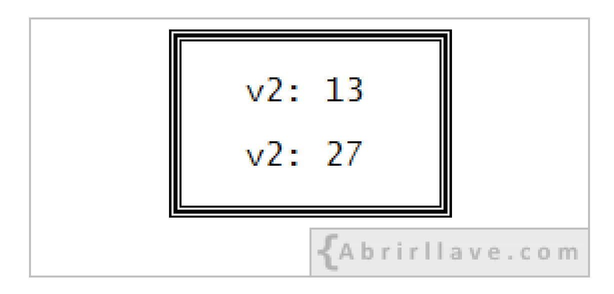
```
algoritmo Ejemplo_legible

constantes
   c1 = 3, c2 = 2, c3 = 4

variables
   entero v1 = 5, v2 = 7

inicio
   v2 ← v2 + c1 * c2
   escribir( "v2: ", v2 )
   v2 ← v1 * c2 * c3 - v2
   escribir( "v2: ", v2 )
fin
```

Obsérvese que, sangrar –tabular– el código de un algoritmo ayuda mucho a su lectura. Así, el código de este algoritmo es mucho más fácil de comprender que el del anterior, siendo la salida por pantalla la misma en ambos casos:



Ejercicios resueltos
• Ejercicios de estructura de un algoritmo en pseudocódigo

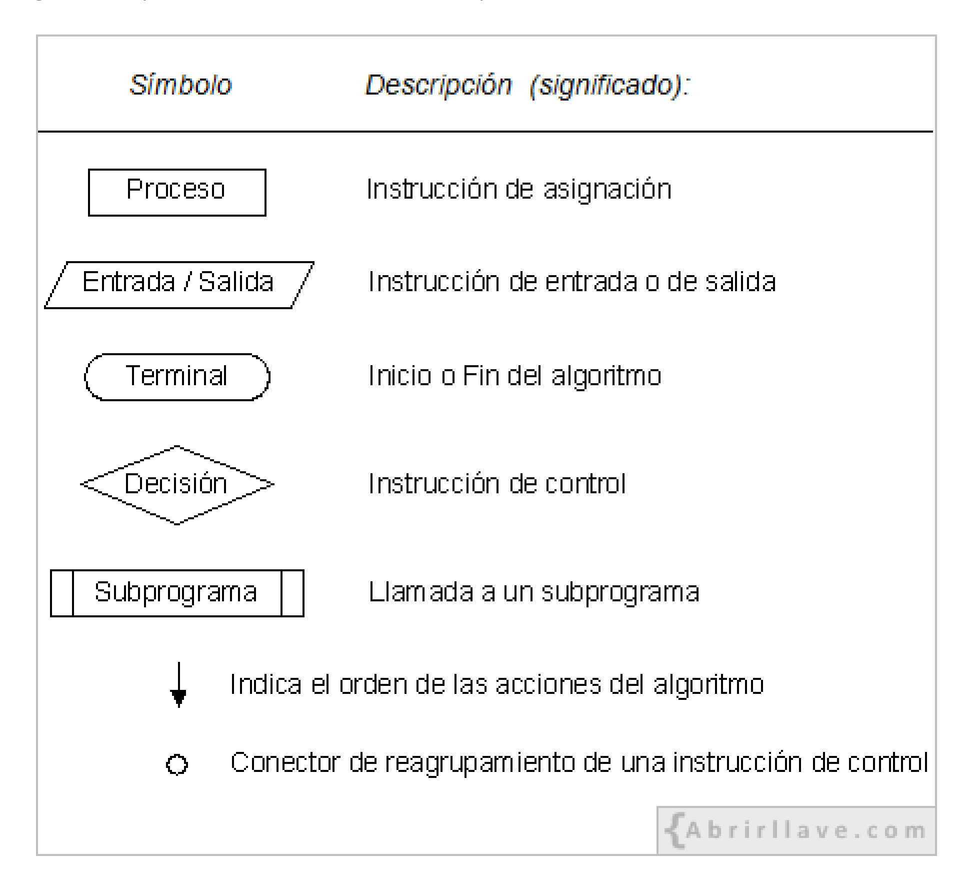
Capítulo 8

Ordinogramas

En este capítulo se va a estudiar cómo es posible representar algoritmos, gráficamente, por medio de *diagramas de flujo*, también llamados **ordinogramas**.

Un **ordinograma** representa, gráficamente, el orden de los “pasos” (acciones) de un algoritmo.

Para representar algoritmos mediante diagramas de flujo, se utilizan una serie de símbolos gráficos que han sido estandarizados por ANSI (*American National Standards Institute*):



8.1. Asignación

Para representar una instrucción de asignación en un ordinograma, se debe escribir la misma sintaxis que en pseudocódigo, pero, dentro de un rectángulo:

```
<nombre_de_la_variable> ← <expresión>
```


EJEMPLO Una instrucción de asignación puede ser:

```
radio ← 5.78
```


EJEMPLO Varias instrucciones de asignación se pueden agrupar dentro de un mismo rectángulo:

```
radio ← 5.78

area ← PI * radio ** 2

longitud ← 2 * PI * radio
```


8.2. Entrada y salida

En un ordinograma, tanto las instrucciones de entrada como las de salida, se escriben igual que en pseudocódigo, pero, dentro de un romboide:

```
leer( <lista_de_variables> )

escribir( <lista_de_expresiones> )
```


EJEMPLO Una instrucción de entrada que lea la variable `radio`, se escribe:

```
leer( radio )
```

EJEMPLO Varias instrucciones de entrada o de salida pueden dibujarse dentro del mismo romboide:

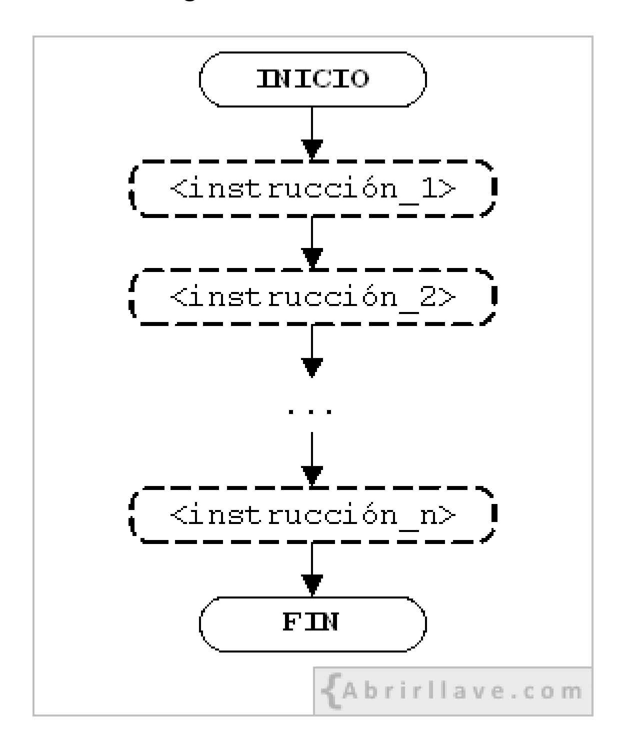
```
escribir( "El área es: ", area )

escribir( "La longitud es: ", longitud )
```

{Abrirllave.com

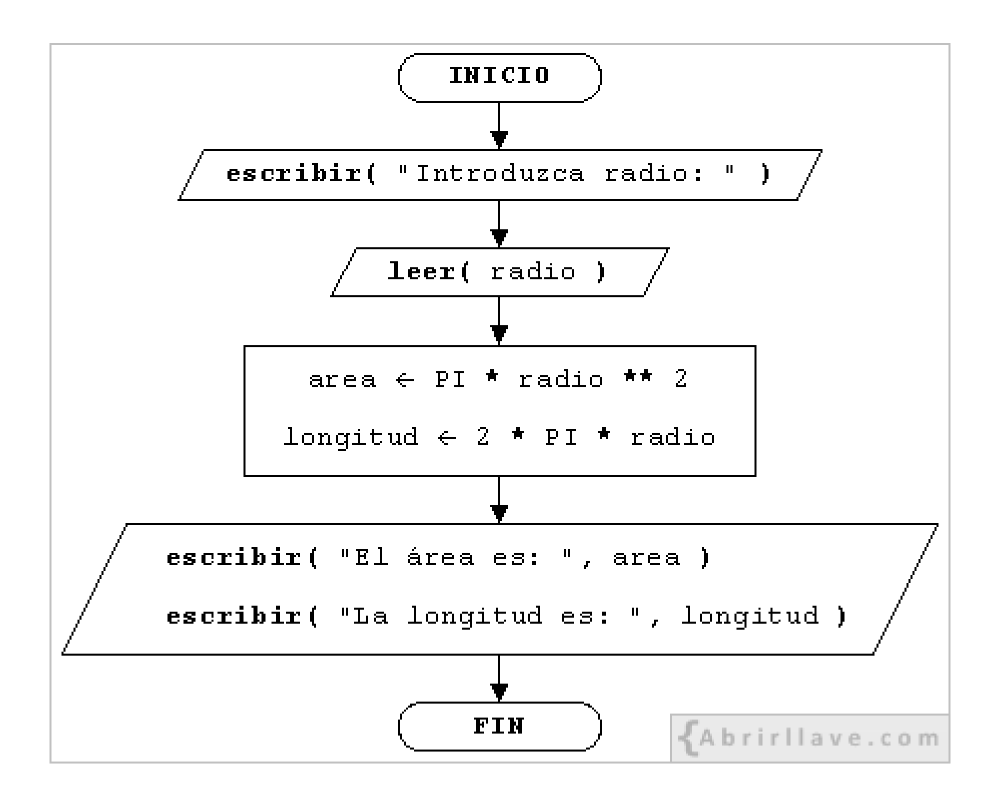
8.3. Inicio y fin

En un ordinograma, el **inicio** y **fin** del cuerpo de un algoritmo se escriben dentro de un óvalo de la siguiente manera:



Por medio de las flechas se indica el orden de las acciones –instrucciones– del algoritmo.

EJEMPLO Así pues, el siguiente ordinograma es equivalente al cuerpo de un algoritmo escrito en pseudocódigo:

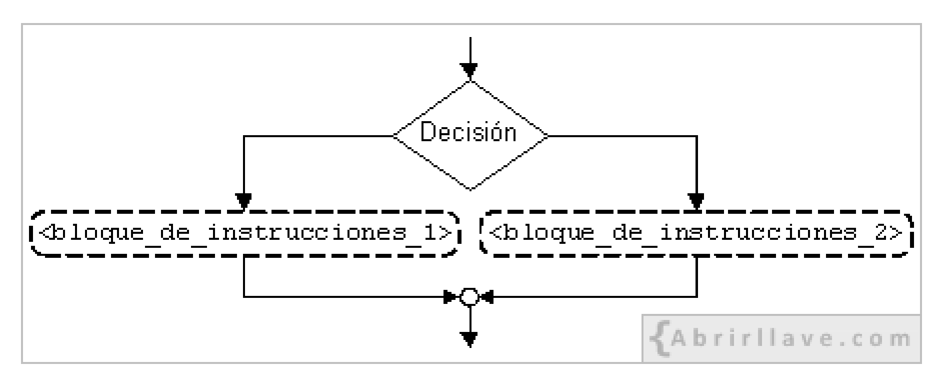


8.4. Decisiones

Como veremos más adelante, las **decisiones** siempre forman parte de las *instrucciones de control*, las cuales sirven para determinar el orden en el que se tienen que ejecutar las instrucciones de un programa.

En un ordinograma, para representar –gráficamente– a una instrucción de control, se utiliza un rombo y un círculo.

EJEMPLO Una *alternativa doble* es una instrucción de control que se representa de la siguiente manera:



En el rombo se toma la decisión de ejecutar un bloque de instrucciones u otro. No obstante, con independencia de cuál de ellos se ejecute, el círculo reagrupa el flujo de control, es decir, la ejecución continuará con la siguiente instrucción que haya después del círculo.

Ejercicios resueltos
• Ejercicios de ordinogramas

Capítulo 9

Instrucciones de control alternativas

Como ya se vio en el capítulo 6 “Instrucciones primitivas”, en programación, las instrucciones que se utilizan para diseñar algoritmos se pueden clasificar en:

- Primitivas.
- De control.
- Llamadas a subalgoritmos (llamadas a subprogramas).

En este capítulo –y en los dos próximos– se van a explicar las instrucciones de control, las cuales se clasifican en:

- Alternativas (selectivas).
- Repetitivas (iterativas).
- De salto (de transferencia).

Flujo de control

Se llama **flujo de control** al orden en el que se ejecutan las instrucciones de un programa, siendo las propias instrucciones las que determinan o controlan dicho flujo.

En un programa, a menos que el flujo de control se vea modificado por una instrucción de control, las instrucciones siempre se ejecutan secuencialmente, una detrás de otra, en orden de aparición, de izquierda a derecha y de arriba abajo, que es el flujo natural de un programa.

En programación estructurada, se considera una mala práctica hacer uso de las instrucciones de salto, ya que, entre otras cosas, restan legibilidad al algoritmo.

Si bien, se debe evitar el uso de las instrucciones de salto, muchos lenguajes de programación, entre ellos C, permiten codificarlas. Por esta razón, las estudiaremos más adelante en este tutorial.

9.1. Instrucciones alternativas

Una **instrucción de control alternativa** permite seleccionar, en el flujo de control de un programa, la o las siguientes instrucciones a ejecutar, de entre varias posibilidades.

Para comprender el porqué son necesarias las instrucciones alternativas, estúdiese el siguiente problema.

EJEMPLO Calificación según nota (Versión 1).

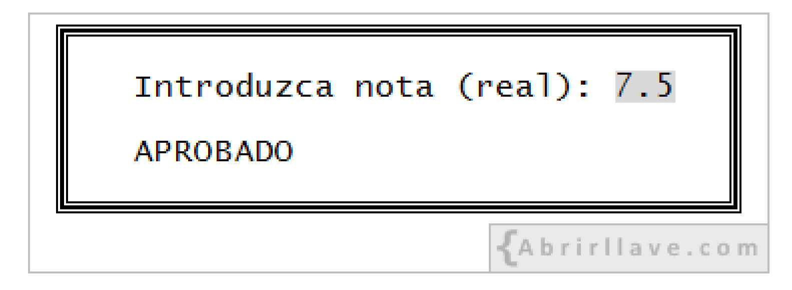
Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado la nota (dato real) de una asignatura.

2º) Muestre por pantalla:

- "APROBADO", en el caso de que la nota sea mayor o igual que 5.
- "SUSPENDIDO", en el caso de que la nota sea menor que 5.

De modo que, por pantalla se verá, por ejemplo:



Otra posibilidad es:

```
Introduzca nota (real): 3.5

SUSPENDIDO
```

Abrirllave.com

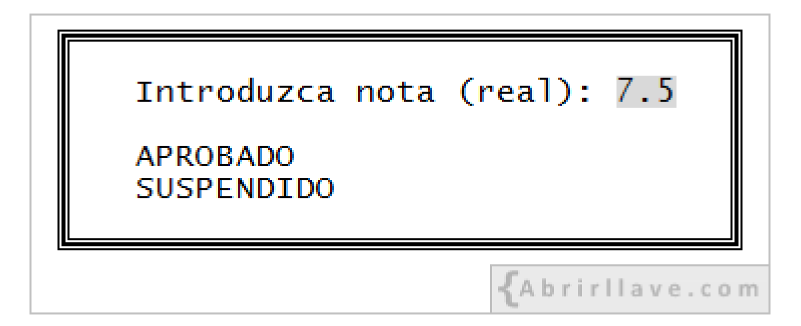
Con lo estudiado hasta ahora, para resolver el problema planteado se puede escribir el siguiente algoritmo erróneo:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )
   escribir( "APROBADO" )
   escribir( "SUSPENDIDO" )
fin
```

Pero, ¿qué sucede? Tal y como se ha escrito, ¡no funciona bien! Se ha cometido un error lógico. Por pantalla se mostrará, por ejemplo:



O también:

```
Introduzca nota (real): 3.5

APROBADO
SUSPENDIDO
```

Abrirllave.com

Después de leer la nota por teclado, se necesita poder seleccionar la siguiente instrucción a ejecutar. Una de entre:

```
escribir( "APROBADO" )
escribir( "SUSPENDIDO" )
```

No deben ejecutarse las dos instrucciones anteriores, sino solamente una de ellas. Una instrucción alternativa va a permitir seleccionar cuál de ellas debe ejecutarse. Existen tres tipos de instrucciones alternativas: doble, simple y múltiple.

Como vamos a ver seguidamente, el problema planteado se puede resolver utilizando una instrucción alternativa doble.

9.1.1. Alternativa doble

En pseudocódigo, para escribir una instrucción alternativa doble se utiliza la sintaxis:

```
si ( <expresión_lógica> )
   <bloque_de_instrucciones_1>
sino
   <bloque_de_instrucciones_2>
fin_si

/* Si la <expresión_lógica> es verdadera,
   se ejecuta el <bloque_de_instrucciones_1>, sino
   se ejecuta el <bloque_de_instrucciones_2>. */
```

A la **<expresión_lógica>** de una instrucción alternativa doble también se le denomina **condición**.

Para que se ejecute el **<bloque_de_instrucciones_1>**, la condición tiene que ser **verdadera**. Por el contrario, si la condición es **falsa**, se ejecutará el **<bloque_de_instrucciones_2>**.

En resumen, una **instrucción alternativa doble** (o simplemente **alternativa doble**) permite seleccionar, por medio de una condición, el siguiente bloque de instrucciones a ejecutar, de entre dos posibles.

EJEMPLO Así pues, el error lógico del ejemplo anterior, se puede resolver con el código:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   si ( nota >= 5 )
      escribir( "APROBADO" )
   sino
      escribir( "SUSPENDIDO" )
   fin_si
fin
```

Este algoritmo sí cumple con las especificaciones del problema, o dicho de otro modo, sí hace lo que se espera de él. Del resultado de evaluar la expresión lógica:

```
nota >= 5
```

Depende que se ejecute la instrucción:

```
escribir( "APROBADO" )
```

O, por el contrario, la instrucción:

```
escribir( "SUSPENDIDO" )
```

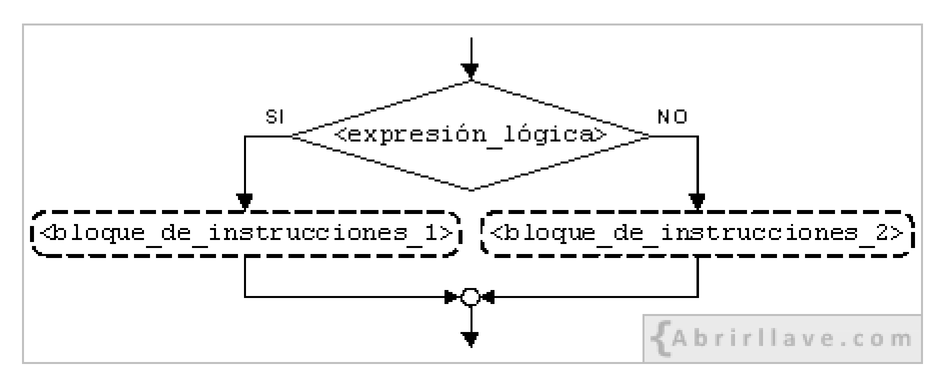
Cuando en un algoritmo existe una condición de la cual depende que a continuación se ejecuten unas instrucciones u otras, se dice que existe una ***bifurcación***.

Han aparecido tres nuevas palabras reservadas: **si**, **sino** y **fin_si**.

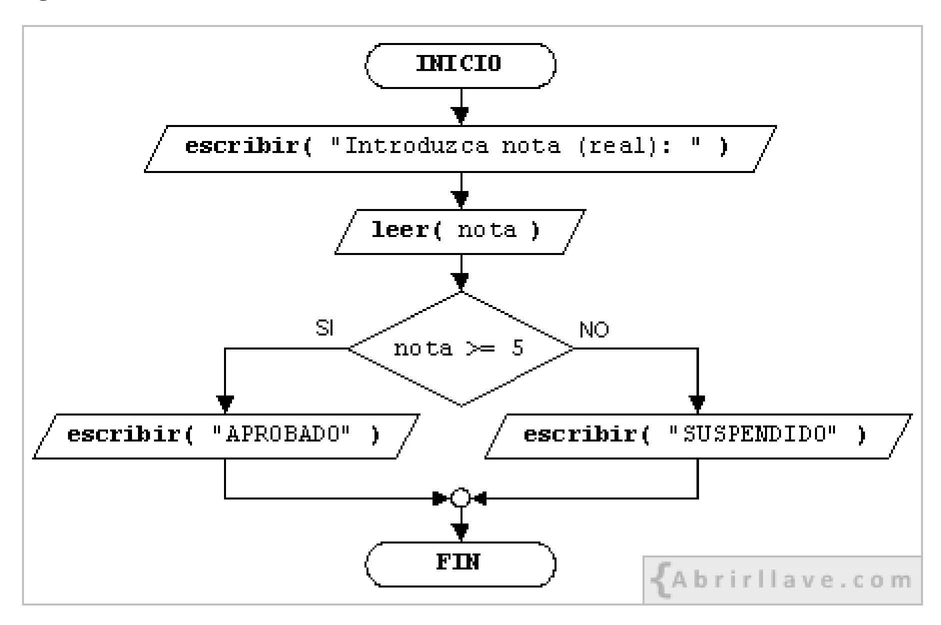
Además, han vuelto a aparecer los símbolos reservados *abrir paréntesis* “**(**” y *cerrar paréntesis* “**)**”, con un significado distinto.

()	Delimitan la condición en una instrucción alternativa doble.

En un ordinograma, una instrucción alternativa doble se representa de la siguiente manera:



En consecuencia, el algoritmo del ejemplo anterior se puede representar, gráficamente, de la siguiente forma:



Ejercicios resueltos
• Ejercicios de la instrucción alternativa doble

9.1.2. Alternativa simple

Una **instrucción alternativa simple** (o simplemente **alternativa simple**) es una variante (más sencilla) de una instrucción alternativa doble. En pseudocódigo, para escribir una alternativa simple se utiliza la sintaxis:

```
si ( <expresión_lógica> )
   <bloque_de_instrucciones>
fin_si

/* Si la <expresión_lógica> es verdadera,
   se ejecuta el <bloque_de_instrucciones>, sino
   no se ejecuta nada. */
```

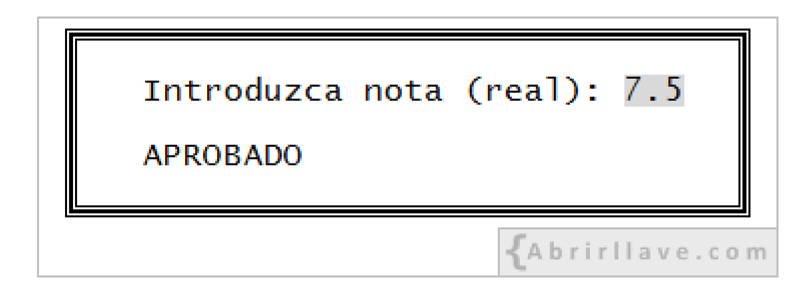
EJEMPLO Calificación según nota (Versión 2).

Se quiere diseñar el algoritmo de un programa que:

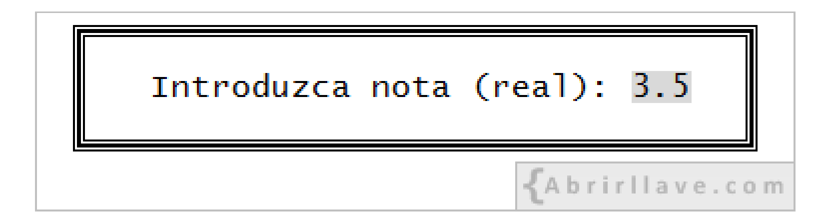
1º) Pida por teclado la nota (dato real) de una asignatura.

2º) Muestre por pantalla:

- "APROBADO", en el caso de que la nota sea mayor o igual que 5.



Obsérvese que, en este problema, no se va a mostrar por pantalla "SUSPENDIDO" en el caso de que la nota sea menor que 5, como sí se hacía en el problema del ejemplo anterior.



El algoritmo puede ser:

```
algoritmo Calificacion_segun_nota

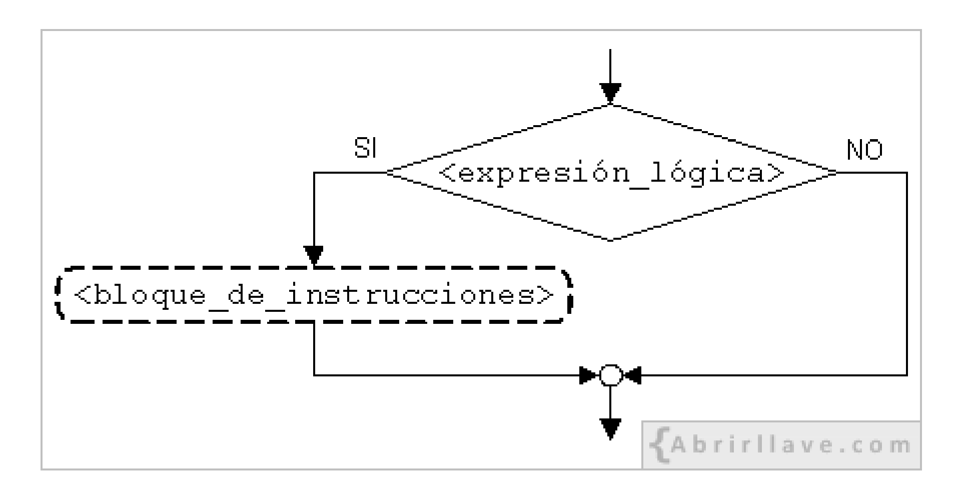
variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

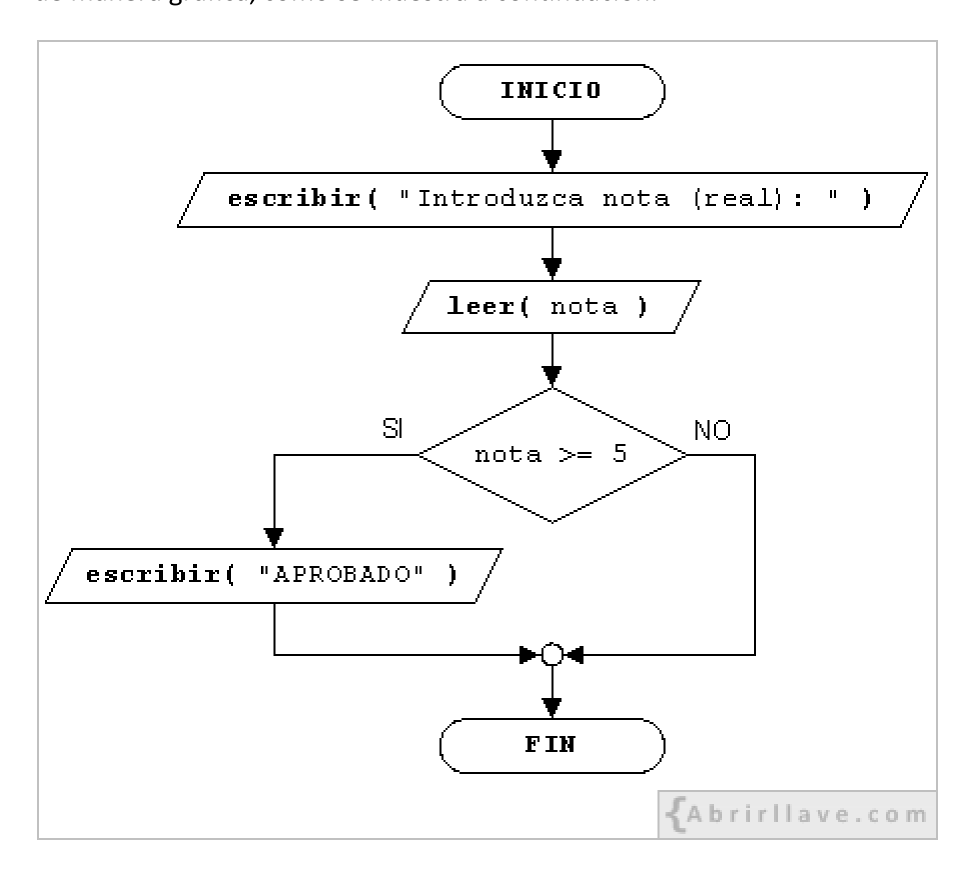
   si ( nota >= 5 )
      escribir( "APROBADO" )
   fin_si
fin
```

En el caso de que la condición (**nota >= 5**) sea **falsa**, el programa terminará, no se ejecutará ninguna instrucción más.

En un ordinograma, una instrucción alternativa simple se representa de la siguiente manera:



Por tanto, el algoritmo del ejemplo "Calificación según nota (Versión 2)" se puede representar, de manera gráfica, como se muestra a continuación:



9.1.3. Alternativa múltiple

Una **instrucción alternativa múltiple** (o simplemente **alternativa múltiple**) permite seleccionar, por medio de una expresión, el siguiente bloque de instrucciones a ejecutar de entre varios posibles. En pseudocódigo, para escribir una alternativa múltiple se utiliza la sintaxis:

```
segun_sea ( <expresión> )
   <lista_de_valores_1> : <bloque_de_instrucciones_1>
   <lista_de_valores_2> : <bloque_de_instrucciones_2>
   ...
   <lista_de_valores_n> : <bloque_de_instrucciones_n>
                 [ sino : <bloque_de_instrucciones_n+1> ]
fin_segun_sea

/* Según sea el valor de evaluar la <expresión>,
   se ejecuta un <bloque_de_instrucciones> u otro. */
```

El resultado de evaluar la **<expresión>** debe ser un valor perteneciente a un tipo de dato finito y ordenado, es decir, entero, lógico, carácter, enumerado o subrango.

Dependiendo del valor obtenido al evaluar la **<expresión>**, se ejecutará un bloque de instrucciones u otro. En las listas de valores se deben escribir los valores que determinan el bloque de instrucciones a ejecutar, teniendo en cuenta que, un valor solamente puede aparecer en una lista de valores.

Opcionalmente, se puede escribir un **<bloque_de_instrucciones_n+1>** después de **sino** :. Este bloque de instrucciones se ejecutará en el caso de que el valor obtenido al evaluar la **<expresión>**, no se encuentre en ninguna de las listas de valores especificadas.

EJEMPLO Día de la semana.

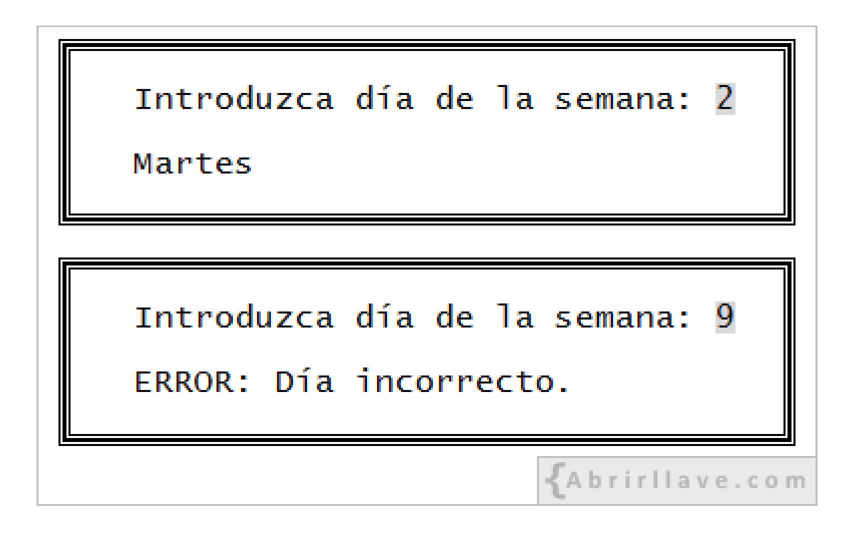
Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado el número (dato entero) de un día de la semana.

2º) Muestre por pantalla el nombre (dato cadena) correspondiente a dicho día.

Nota: Si el número de día introducido es menor que 1 ó mayor que 7, se mostrará el mensaje: "ERROR: Día incorrecto.".

En pantalla:



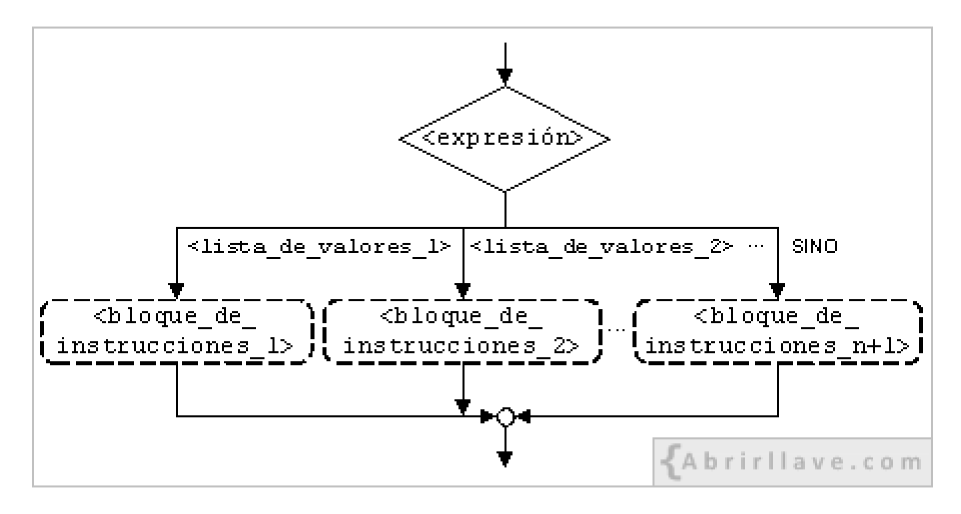
Algoritmo propuesto:

```
algoritmo Dia_de_la_semana

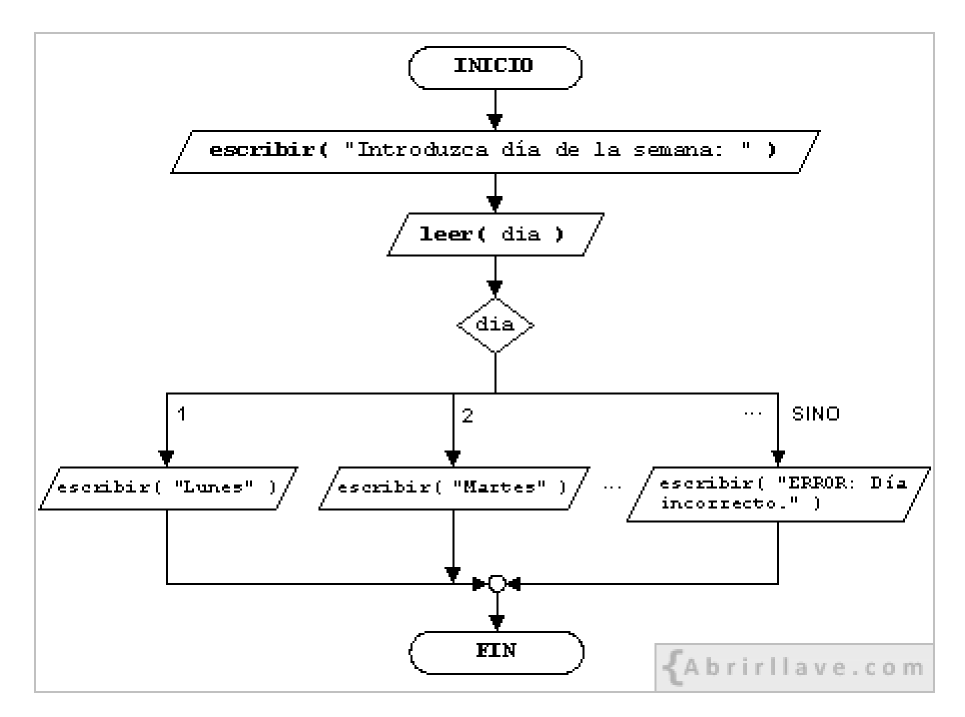
variables
   entero dia

inicio
   escribir( "Introduzca día de la semana: " )
   leer( dia )
   segun_sea ( dia )
         1 : escribir( "Lunes" )
         2 : escribir( "Martes" )
         3 : escribir( "Miércoles" )
         4 : escribir( "Jueves" )
         5 : escribir( "Viernes" )
         6 : escribir( "Sábado" )
         7 : escribir( "Domingo" )
      sino : escribir( "ERROR: Día incorrecto." )
   fin_segun_sea
fin
```

En un ordinograma, una instrucción alternativa múltiple se representa del siguiente modo:



Por tanto, el algoritmo del ejemplo "Día de la semana" se puede representar, de manera gráfica, de la siguiente forma:



Cuando en una lista de valores de una alternativa múltiple aparece más de un valor, estos se escriben separados por el carácter coma (**,**). Estúdiese el siguiente ejemplo.

EJEMPLO Signo del zodíaco.

En la siguiente tabla se muestran las categorías a las que pertenecen los signos del zodíaco:

Signo	Categoría
1. Aries	Fuego
2. Tauro	Tierra
3. Géminis	Aire
4. Cáncer	Agua
5. Leo	Fuego
6. Virgo	Tierra
7. Libra	Aire
8. Escorpio	Agua
9. Sagitario	Fuego
10. Capricornio	Tierra
11. Acuario	Aire
12. Piscis	Agua

Se quiere diseñar el algoritmo de un programa que:

1º) Muestre el listado de los signos del zodíaco, con sus números asociados.

2º) Pida por teclado un número (dato entero) asociado a un signo del zodíaco.

3º) Muestre la categoría a la que pertenece el signo del zodíaco seleccionado.

Nota: Si el número introducido por el usuario, no está asociado a ningún signo del zodíaco, se mostrará el mensaje: "ERROR: <número> no está asociado a ningún signo.".

En pantalla:

```
Listado de signos del zodíaco:

1. Aries
2. Tauro
3. Géminis
4. Cáncer
5. Leo
6. Virgo
7. Libra
8. Escorpio
9. Sagitario
10. Capricornio
11. Acuario
12. Piscis

Introduzca número de signo: 7

Es un signo de Aire.
```



```
Listado de signos del zodíaco:

1. Aries
2. Tauro
3. Géminis
4. Cáncer
5. Leo
6. Virgo
7. Libra
8. Escorpio
9. Sagitario
10. Capricornio
11. Acuario
12. Piscis

Introduzca número de signo: 15

ERROR: 15 no está asociado a ningún signo.
```

Una posible solución es:

```
algoritmo Signo_del_zodiaco

variables
   entero numero

inicio
   escribir( "Listado de signos del zodíaco:" )
   escribir( "1. Aries" )
   escribir( "2. Tauro" )
   escribir( "3. Géminis" )
   escribir( "4. Cáncer" )
   escribir( "5. Leo" )
   escribir( "6. Virgo" )
   escribir( "7. Libra" )
   escribir( "8. Escorpio" )
   escribir( "9. Sagitario" )
   escribir( "10. Capricornio" )
   escribir( "11. Acuario" )
   escribir( "12. Piscis" )
   escribir( "Introduzca número de signo: " )

   leer( numero )

   segun_sea ( numero )
       1, 5, 9 : escribir( "Es un signo de Fuego." )
      2, 6, 10 : escribir( "Es un signo de Tierra." )
      3, 7, 11 : escribir( "Es un signo de Aire." )
      4, 8, 12 : escribir( "Es un signo de Agua." )
          sino : escribir( "ERROR: ", numero,
                 " no está asociado a ningún signo." )
   fin_segun_sea
fin
```

Otra solución es:

```
algoritmo Signo_del_zodiaco

variables
   entero numero
   cadena categoria

inicio
   escribir( "Listado de signos del zodíaco:" )
   escribir( "1. Aries" )
   escribir( "2. Tauro" )
   escribir( "3. Géminis" )
   escribir( "4. Cáncer" )
   escribir( "5. Leo" )
   escribir( "6. Virgo" )
   escribir( "7. Libra" )
   escribir( "8. Escorpio" )
   escribir( "9. Sagitario" )
   escribir( "10. Capricornio" )
   escribir( "11. Acuario" )
   escribir( "12. Piscis" )
   escribir( "Introduzca número de signo: " )

   leer( numero )

   segun_sea ( numero mod 4 )
      1 : categoria ← "Fuego"
      2 : categoria ← "Tierra"
      3 : categoria ← "Aire"
      0 : categoria ← "Agua"
   fin_segun_sea

   si ( numero >= 1 y numero <= 12 )
      escribir( "Es un signo de ", categoria, "." )
   sino
      escribir( "ERROR: ", numero,
      " no está asociado a ningún signo." )
   fin_si
fin
```

En esta segunda solución existen las siguientes diferencias importantes con respecto a la solución anterior:

- En el algoritmo se utiliza una alternativa doble, además de una alternativa múltiple.
- En la alternativa múltiple no se escribe el **<bloque_de_instrucciones _n+1>**.
- La expresión de la alternativa múltiple es diferente.
- La expresión "**Es un signo de** " solamente se escribe una vez.
- Se ha utilizado una variable más: **categoria**

Han vuelto a aparecer los símbolos reservados *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**", con un significado distinto:

()	Delimitan la expresión en una instrucción alternativa múltiple.

Y los símbolos *coma* (`,`) y *dos puntos* (`:`):

`,`	Separadora de los valores de una lista de valores de una instrucción alternativa múltiple.
`:`	Separador de una lista de valores y de un bloque de instrucciones, en una instrucción alternativa múltiple.

También han aparecido dos nuevas palabras reservadas: **`segun_sea`** y **`fin_segun_sea`**. Además de usarse de nuevo la palabra reservada **`sino`**.

Ejercicios resueltos
• Ejercicios de la instrucción alternativa múltiple

9.2. Anidamiento de alternativas

Las instrucciones alternativas y repetitivas pueden escribirse una dentro de otra. A este hecho se le conoce como **anidamiento**.

En este apartado del tutorial se estudia el anidamiento de instrucciones alternativas, mientras que, el anidamiento de instrucciones repetitivas se va a tratar en el apartado siguiente.

Las instrucciones alternativas permiten realizar las siguientes combinaciones de anidamiento:

- Doble en doble.
- Doble en simple.
- Doble en múltiple.
- Simple en simple.
- Simple en doble.
- Simple en múltiple.
- Múltiple en múltiple.
- Múltiple en doble.
- Múltiple en simple.

De ellas, vamos a estudiar, como ejemplo, las siguientes combinaciones:

- Doble en doble.
- Múltiple en doble.

9.2.1. Alternativa doble en doble

En pseudocódigo, para anidar una alternativa doble en otra, se utiliza la sintaxis:

```
si ( <expresión_lógica_1> )

   /* Inicio del anidamiento */
   si ( <expresión_lógica_2> )
      <bloque_de_instrucciones_1>
   sino
      <bloque_de_instrucciones_2>
   fin_si
   /* Fin del anidamiento */

sino
   <bloque_de_instrucciones_3>
fin_si
```

O también:

```
si ( <expresión_lógica_1> )
   <bloque_de_instrucciones_1>
sino

   /* Inicio del anidamiento */
   si ( <expresión_lógica_2> )
      <bloque_de_instrucciones_2>
   sino
      <bloque_de_instrucciones_3>
   fin_si
   /* Fin del anidamiento */

fin_si
```

EJEMPLO Calificación según nota (Versión 3).

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado la nota (dato real) de una asignatura.

2º) Muestre por pantalla:

- “APTO”, en el caso de que la nota sea mayor o igual que 5 y menor o igual que 10.
- “NO APTO”, en el caso de que la nota sea mayor o igual que 0 y menor que 5.
- “ERROR: Nota incorrecta.”, en el caso de que la nota sea menor que 0 o mayor que 10.

En pantalla:

```
Introduzca nota (real): 7.5
APTO
```

```
Introduzca nota (real): 12.2
ERROR: Nota incorrecta.
```


Una solución al problema es:

```
algoritmo Calificacion_segun_nota

variables
   real nota

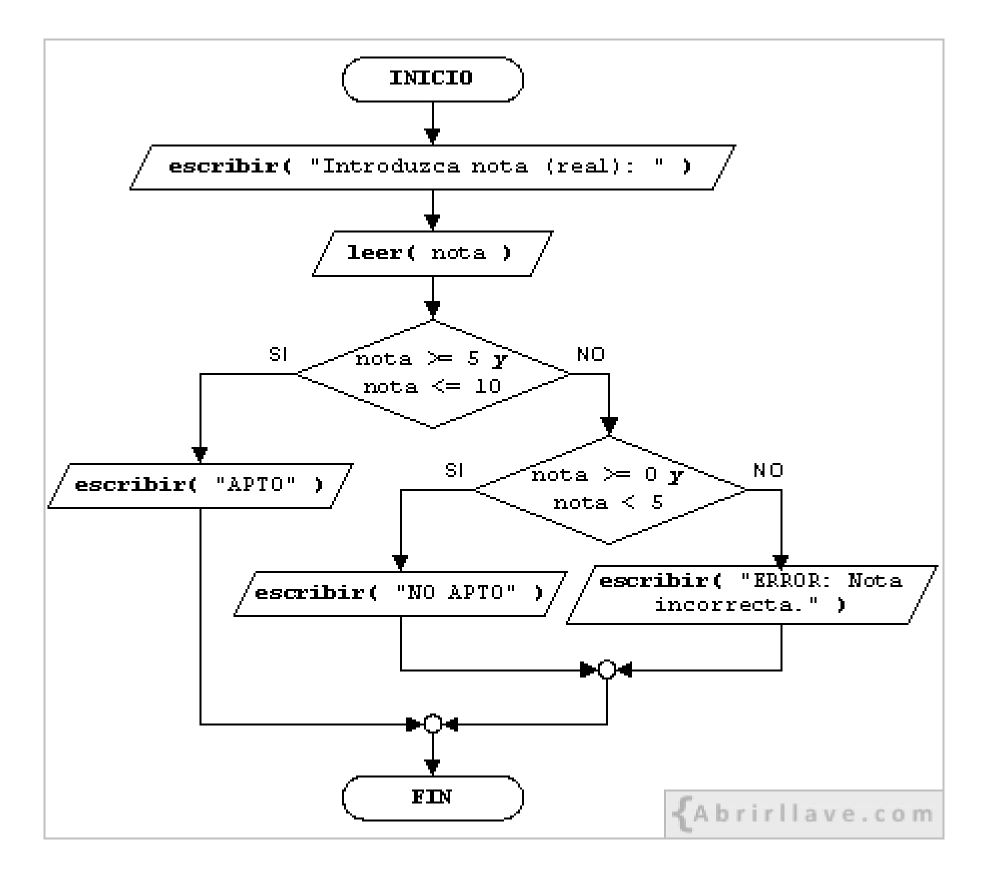
inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   si ( nota >= 5 y nota <= 10 )
      escribir( "APTO" )
   sino

      /* Inicio del anidamiento */
      si ( nota >= 0 y nota < 5 )
         escribir( "NO APTO" )
      sino
         escribir( "ERROR: Nota incorrecta." )
      fin_si
      /* Fin del anidamiento */

   fin_si
fin
```

En ordinograma:



Una segunda solución es:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   si ( nota < 0 o nota > 10 )
      escribir( "ERROR: Nota incorrecta." )
   sino

      /* Inicio del anidamiento */
      si ( nota < 5 )
         escribir( "NO APTO" )
      sino
         escribir( "APTO" )
      fin_si
      /* Fin del anidamiento */

   fin_si
fin
```

Una tercera solución es:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   si ( nota >= 0 y nota <= 10 )

      /* Inicio del anidamiento */
      si ( nota >= 5 )
         escribir( "APTO" )
      sino
         escribir( "NO APTO" )
      fin_si
      /* Fin del anidamiento */

   sino
      escribir( "ERROR: Nota incorrecta." )
   fin_si
fin
```

Como se puede observar, el anidamiento de instrucciones alternativas permite ir descartando valores hasta llegar al bloque de instrucciones que se debe ejecutar. Estúdiese ahora el siguiente problema, el cual es un poco más complejo.

EJEMPLO Calificación según nota (Versión 4).

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado la nota (dato real) de una asignatura.

2º) Muestre por pantalla:

- "SOBRESALIENTE", en el caso de que la nota sea mayor o igual que 9 y menor o igual que 10.
- "NOTABLE", en el caso de que la nota sea mayor o igual que 7 y menor que 9.
- "BIEN", en el caso de que la nota sea mayor o igual que 6 y menor que 7.
- "SUFICIENTE", en el caso de que la nota sea mayor o igual que 5 y menor que 6.
- "INSUFICIENTE", en el caso de que la nota sea mayor o igual que 3 y menor que 5.
- "MUY DEFICIENTE", en el caso de que la nota sea mayor o igual que 0 y menor que 3.
- "ERROR: Nota incorrecta.", en el caso de que la nota sea menor que 0 o mayor que 10.

En pantalla:

```
Introduzca nota (real): 7.5

NOTABLE
```

```
Introduzca nota (real): -3.3

ERROR: Nota incorrecta.
```


La solución propuesta tiene más de un nivel de anidamiento:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )
   si ( nota < 0 o nota > 10 )
      escribir( "ERROR: Nota incorrecta." )
   sino
      si ( nota >= 9 )
         escribir( "SOBRESALIENTE" )
      sino
         si ( nota >= 7 )
            escribir( "NOTABLE" )
         sino
            si ( nota >= 6 )
               escribir( "BIEN" )
            sino
               si ( nota >= 5 )
                  escribir( "SUFICIENTE" )
               sino
                  si ( nota >= 3 )
                     escribir( "INSUFICIENTE" )
                  sino
                     escribir( "MUY DEFICIENTE" )
                  fin_si
               fin_si
            fin_si
         fin_si
      fin_si
   fin_si
fin
```

9.2.2. Alternativa múltiple en doble

En pseudocódigo, para anidar una alternativa múltiple en una alternativa doble, se utiliza la sintaxis:

```
si ( <expresión_lógica> )

   /* Inicio del anidamiento */
   segun_sea ( <expresión> )
      <lista_de_valores_1> : <bloque_de_instrucciones_1>
      <lista_de_valores_2> : <bloque_de_instrucciones_2>
      ...
      <lista_de_valores_n> : <bloque_de_instrucciones_n>
                     [ sino : <bloque_de_instrucciones_n+1> ]
   fin_segun_sea
   /* Fin del anidamiento */

sino
   <bloque_de_instrucciones_n+2>
fin_si
```

Así por ejemplo, el problema del "Día de la semana" visto anteriormente, también se puede resolver anidando una alternativa múltiple en una alternativa doble.

EJEMPLO Día de la semana.

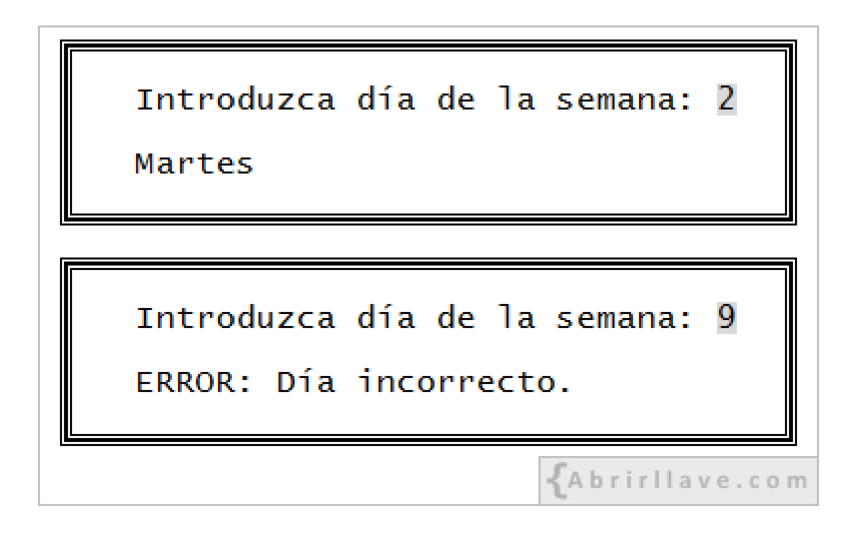
Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado el número (dato entero) de un día de la semana.

2º) Muestre por pantalla el nombre (dato cadena) correspondiente a dicho día.

Nota: Si el número de día introducido es menor que 1 ó mayor que 7, se mostrará el mensaje: "ERROR: Día incorrecto.".

En pantalla:



La solución algorítmica podría ser:

```
algoritmo Dia_de_la_semana

variables
   entero dia

inicio
   escribir( "Introduzca día de la semana: " )
   leer( dia )
   si ( dia >= 1 y dia <= 7 )

   /* Solamente si el día es válido, se ejecuta la
   instrucción alternativa múltiple. */

      /* Inicio del anidamiento */
      segun_sea ( dia )
         1 : escribir( "Lunes" )
         2 : escribir( "Martes" )
         3 : escribir( "Miércoles" )
         4 : escribir( "Jueves" )
         5 : escribir( "Viernes" )
         6 : escribir( "Sábado" )
         7 : escribir( "Domingo" )
      fin_segun_sea
      /* Fin del anidamiento */

   sino
      escribir ( "ERROR: Día incorrecto." )
   fin_si
fin
```

En este algoritmo, se ha comprobado –en primer lugar– el valor del día y, solamente se ejecutará la alternativa múltiple, si el valor introducido es mayor o igual que 1 y menor o igual que 7.

9.3. Distintas soluciones para un problema

En programación, para solucionar un problema, se pueden diseñar algoritmos distintos, o dicho de otro modo, no existe una única solución para resolver un problema dado. Pero, a veces, unas soluciones son mejores –más óptimas– que otras.

Cuando se dice que un algoritmo es mejor –más óptimo– que otro, teniendo ambos la misma funcionalidad, esto puede ser debido a distintas razones. Entre ellas, de momento, vamos a destacar dos:

- El código es más reducido (se ejecutan menos instrucciones).
- Utiliza menos variables (menos memoria).

EJEMPLO Calificación según nota (Versión 5).

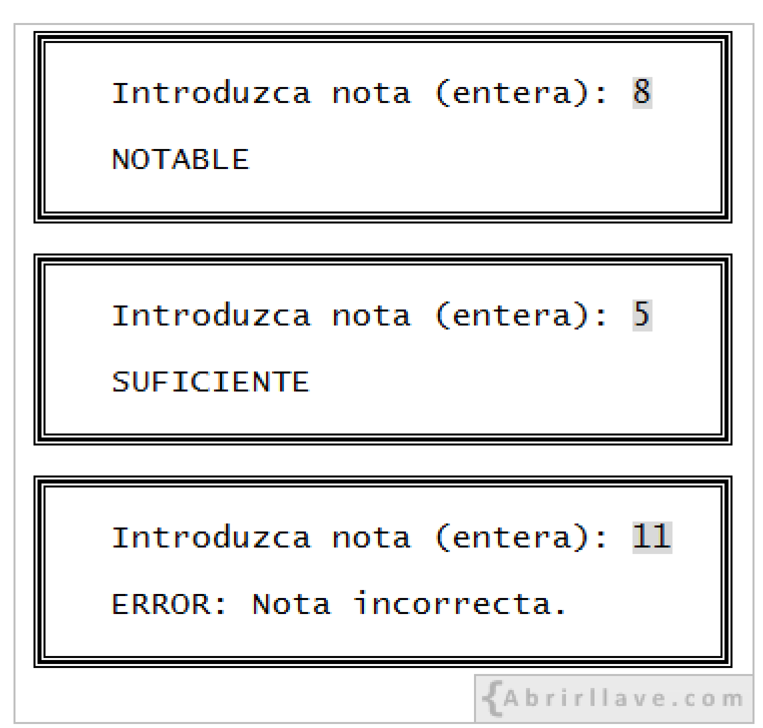
Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado la nota (dato entero) de una asignatura.

2º) Muestre por pantalla:

- "SOBRESALIENTE", en el caso de que la nota sea un 9 ó un 10.
- "NOTABLE", en el caso de que la nota sea un 7 ó un 8.
- "BIEN", en el caso de que la nota sea un 6.
- "SUFICIENTE", en el caso de que la nota sea un 5.
- "INSUFICIENTE", en el caso de que la nota sea un 3 ó un 4.
- "MUY DEFICIENTE", en el caso de que la nota sea un 0, un 1, ó un 2.
- "ERROR: Nota incorrecta.", en el caso de que la nota sea menor que 0 o mayor que 10.

En pantalla:



Para resolver este problema, se puede recurrir al anidamiento de alternativas dobles:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (entera): " )
   leer( nota )
   si ( nota >= 0 y nota <= 10 )
      si ( nota >= 9 )
         escribir( "SOBRESALIENTE" )
      sino
         si ( nota >= 7 )
            escribir( "NOTABLE" )
         sino
            si ( nota = 6 )
               escribir( "BIEN" )
            sino
               si ( nota = 5 )
                  escribir( "SUFICIENTE" )
               sino
                  si ( nota >= 3 )
                     escribir( "INSUFICIENTE" )
                  sino
                     escribir( "MUY DEFICIENTE" )
                  fin_si
               fin_si
            fin_si
         fin_si
      fin_si
   sino
      escribir( "ERROR: Nota incorrecta." )
   fin_si
fin
```

Otra posibilidad, para resolver este problema, es utilizar una alternativa múltiple anidada en una alternativa doble:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (entera): " )
   leer( nota )

   si ( nota >= 0 y nota <= 10 )
      segun_sea ( nota )
           10, 9 : escribir( "SOBRESALIENTE" )
            8, 7 : escribir( "NOTABLE" )
               6 : escribir( "BIEN" )
               5 : escribir( "SUFICIENTE" )
            4, 3 : escribir( "INSUFICIENTE" )
         2, 1, 0 : escribir( "MUY DEFICIENTE" )
      fin_segun_sea
   sino
      escribir( "ERROR: Nota incorrecta." )
   fin_si
fin
```

Obsérvese que, a primera vista, el algoritmo de la segunda solución es más fácil de leer que el de la primera solución.

Podría pensarse, erróneamente, que el problema del ejemplo "Calificación según nota (Versión 4)" visto en el apartado "9.2. Anidamiento de alternativas" también se puede resolver con una alternativa múltiple, pero no es así, ya que, la nota que se pide por teclado en ese problema, es un dato de tipo real y, como ya se ha estudiado, en una alternativa múltiple, el valor de la expresión que se evalúe debe pertenecer a un tipo de dato finito y ordenado.

9.4. Variable interruptor

Una variable interruptor es un tipo de variable que se utiliza con frecuencia en programación.

Un **interruptor** es una variable que solamente puede tomar por valor dos valores opuestos. Por norma general, estos valores son: **`verdadero`** y **`falso`**. También es frecuente utilizar los valores: **0** y **1**.

Normalmente, una variable interruptor tomará un valor u otro dependiendo de ciertas circunstancias ocurridas en el algoritmo y, después, según sea su valor, se ejecutarán unas instrucciones u otras.

EJEMPLO Validar fecha.

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado una fecha en tres variables: día, mes y año (datos enteros).

2º) Muestre por pantalla:

- "FECHA CORRECTA", en el caso de que la fecha sea válida.
- "FECHA INCORRECTA", en el caso de que la fecha no sea válida.

Nota1: Para que una fecha sea válida, se tiene que cumplir que:

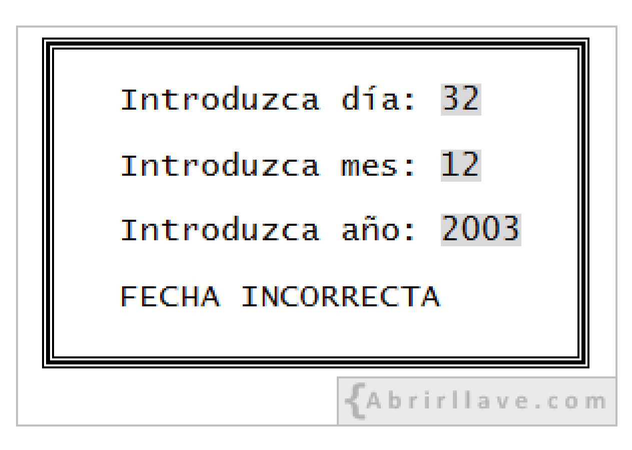
- El mes debe ser mayor o igual que 1 y menor o igual que 12.
- El día debe ser mayor o igual que 1 y menor o igual que un número, el cual dependerá del mes y año introducidos por el usuario.

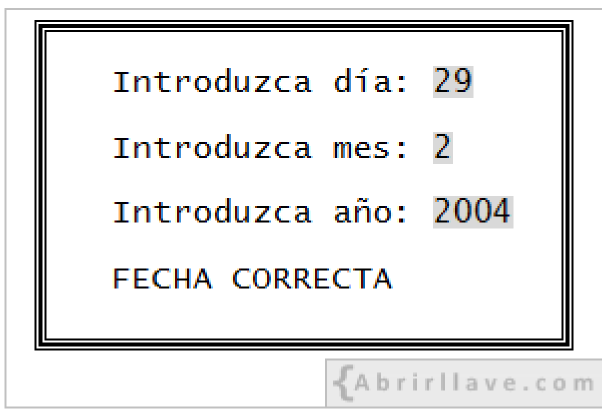
Nota2: Hay que tener en cuenta que:

- Tienen 31 días: enero, marzo, mayo, julio, agosto, octubre y diciembre.
- Tienen 30 días: abril, junio, septiembre y noviembre.
- Tiene 29 días: febrero (si el año es bisiesto).
- Tiene 28 días: febrero (si el año no es bisiesto).

Nota3: Son bisiestos todos los años múltiplos de 4, excepto aquellos que son múltiplos de 100 pero no de 400.

En pantalla:





Una posible solución es:

```
algoritmo Validar_fecha

variables
   entero dia, mes, anio

inicio
   escribir( "Introduzca dia: " )
   leer( dia )
   escribir( "Introduzca mes: " )
   leer( mes )
   escribir( "Introduzca año: " )
   leer( anio )

   si ( mes >= 1 y mes <= 12 )

      segun_sea ( mes )
          1, 3, 5, 7,
          8, 10, 12 : si ( dia >= 1 y dia <= 31 )
                         escribir( "FECHA CORRECTA" )
                      sino
                         escribir( "FECHA INCORRECTA" )
                      fin_si

        4, 6, 9, 11 : si ( dia >= 1 y dia <= 30 )
                         escribir( "FECHA CORRECTA" )
                      sino
                         escribir( "FECHA INCORRECTA" )
                      fin_si

                  2 : si ( anio mod 4 = 0 y
                           anio mod 100 <> 0 o
                           anio mod 400 = 0 )
                         si ( dia >= 1 y dia <= 29 )
                            escribir( "FECHA CORRECTA" )
                         sino
                            escribir( "FECHA INCORRECTA" )
                         fin_si
                      sino
                         si( dia >= 1 y dia <= 28 )
                            escribir( "FECHA CORRECTA" )
                         sino
                            escribir( "FECHA INCORRECTA" )
                         fin_si
                      fin_si
      fin_segun_sea

   sino
      escribir( "FECHA INCORRECTA" )
   fin_si
fin
```

A continuación, se muestra una segunda solución, en la cual, se hace uso de una variable interruptor:

```
algoritmo Validar_fecha

variables
   entero dia, mes, anio
   logico fecha_correcta /* Interruptor */

inicio
   escribir( "Introduzca dia: " )
   leer( dia )
   escribir( "Introduzca mes: " )
   leer( mes )
   escribir( "Introduzca año: " )
   leer( anio )

   fecha_correcta ← falso

   si ( mes >= 1 y mes <= 12 )

      segun_sea ( mes )
          1, 3, 5, 7,
          8, 10, 12 : si ( dia >= 1 y dia <= 31 )
                         fecha_correcta ← verdadero
                      fin_si

        4, 6, 9, 11 : si ( dia >= 1 y dia <= 30 )
                         fecha_correcta ← verdadero
                      fin_si

                  2 : si ( anio mod 4 = 0 y
                           anio mod 100 <> 0 o
                           anio mod 400 = 0 )
                         si ( dia >= 1 y dia <= 29 )
                            fecha_correcta ← verdadero
                         fin_si
                      sino
                         si ( dia >= 1 y dia <= 28 )
                            fecha_correcta ← verdadero
                         fin_si
                      fin_si
      fin_segun_sea

   fin_si

 /* Llegados a este punto, según el valor de fecha_correcta,
    por pantalla se mostrará un mensaje u otro. */
   si ( fecha_correcta )
      escribir( "FECHA CORRECTA" )
   sino
      escribir( "FECHA INCORRECTA" )
   fin_si
fin
```

En el algoritmo, la variable interruptor **fecha_correcta** toma el valor **verdadero** o **falso**, dependiendo de que la fecha sea válida o no. Y después, según sea su valor, se ejecutará la instrucción:

```
escribir( "FECHA CORRECTA" )
```

O la instrucción:

```
escribir( "FECHA INCORRECTA" )
```

A los interruptores también se les denomina: **banderas**, **centinelas** o **conmutadores**.

Por último, se propone una tercera solución:

```
algoritmo Validar_fecha

variables
   entero dia_maximo, dia, mes, anio
   logico fecha_correcta /* Interruptor */

inicio
   escribir( "Introduzca dia: " )
   leer( dia )
   escribir( "Introduzca mes: " )
   leer( mes )
   escribir( "Introduzca año: " )
   leer( anio )

   fecha_correcta ← falso

   si ( mes >= 1 y mes <= 12 )
      segun_sea ( mes )
         1, 3, 5, 7, 8, 10, 12 : dia_maximo ← 31
                   4, 6, 9, 11 : dia_maximo ← 30
                             2 : si ( anio mod 4 = 0 y
                                      anio mod 100 <> 0 o
                                      anio mod 400 = 0 )
                                    dia_maximo ← 29
                                 sino
                                    dia_maximo ← 28
                                 fin_si
      fin_segun_sea

      si ( dia >= 1 y dia <= dia_maximo )
         fecha_correcta ← verdadero
      fin_si
   fin_si

   si ( fecha_correcta )
      escribir( "FECHA CORRECTA" )
   sino
      escribir( "FECHA INCORRECTA" )
   fin_si
fin
```

Ejercicios resueltos
• Ejercicios de instrucciones alternativas

Capítulo 10

Instrucciones de control repetitivas

Como ya se estudió en el capítulo 6 "Instrucciones primitivas", en programación, las instrucciones que se utilizan para diseñar algoritmos se pueden clasificar en:

- Primitivas.
- De control.
- Llamadas a subalgoritmos (llamadas a subprogramas).

A su vez, las instrucciones de control se clasifican en:

- Alternativas (selectivas).
- Repetitivas (iterativas).
- De salto (de transferencia).

Seguidamente, se van a estudiar las instrucciones de control repetitivas.

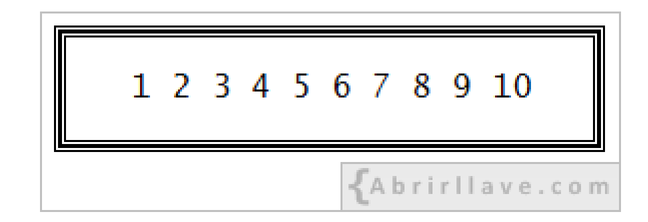
10.1. Instrucciones repetitivas

Una **instrucción de control repetitiva** permite ejecutar una o más instrucciones varias veces.

Para comprender por qué son necesarias las instrucciones repetitivas, también llamadas iterativas, estúdiese el problema del ejemplo siguiente.

EJEMPLO Primeros diez números naturales (del 1 al 10).

Se quiere diseñar el algoritmo de un programa que muestre por pantalla los primeros diez números naturales:



Para resolver el problema planteado, con lo estudiado hasta ahora en este tutorial, se puede escribir el siguiente algoritmo:

```
algoritmo Numeros_del_1_al_10

inicio
   escribir( 1 )
   escribir( 2 )
   escribir( 3 )
   escribir( 4 )
   escribir( 5 )
   escribir( 6 )
   escribir( 7 )
   escribir( 8 )
   escribir( 9 )
   escribir( 10 )
fin
```

El algoritmo funciona bien, pero, ¿y si se quisiera mostrar por pantalla los primeros mil números naturales, o los primeros diez mil? El número de líneas del algoritmo se incrementaría considerablemente.

EJEMPLO Otra posibilidad es utilizar una variable:

```
algoritmo Numeros_del_1_al_10

 variables
    entero numero

 inicio
    numero ← 1
    escribir( numero )      /* Escribe el 1 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 2 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 3 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 4 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 5 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 6 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 7 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 8 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 9 */
    numero ← numero + 1
    escribir( numero )      /* Escribe el 10 */
 fin
```

El número de instrucciones de este algoritmo es todavía mayor que el del anterior. Los dos algoritmos realizan, repetidamente, una determinada tarea. En el primer algoritmo se repite la instrucción de salida:

```
escribir( <número> )
```

En el segundo algoritmo se repite una instrucción de asignación y una instrucción de salida:

```
numero ← numero + 1
escribir( numero )
```

Vamos a ver que, las instrucciones repetitivas permiten ejecutar un bloque de instrucciones repetidamente, escribiéndolas una sola vez en el algoritmo, reduciendo de este modo el código del mismo. Existen tres tipos de instrucciones repetitivas:

- Mientras.
- Hacer...mientras.
- Para.

El problema planteado se puede resolver utilizando, indistintamente, cualquiera de las tres instrucciones repetitivas. Sin embargo, como veremos en el apartado *"10.5. Cuándo usar un bucle u otro"*, para algunas tareas puede ser más conveniente utilizar una determinada instrucción que otra.

A las instrucciones repetitivas también se las conoce como **bucles**, **ciclos** o **lazos**.

10.2. Repetitiva `mientras`

En pseudocódigo, para escribir una instrucción repetitiva **`mientras`**, se utiliza la sintaxis:

```
mientras ( <expresión_lógica> )
   <bloque_de_instrucciones>
fin_mientras

/* Mientras que la <expresión_lógica> sea verdadera,
   se ejecuta el <bloque_de_instrucciones>. */
```

Igual que en las instrucciones alternativas doble y simple, a la **`<expresión_lógica>`** de una instrucción repetitiva **`mientras`**, también se le llama **condición**.

Para que se ejecute el **`<bloque_de_instrucciones>`**, la condición tiene que ser **verdadera**. Por el contrario, si la condición es **falsa**, el **`<bloque_de_instrucciones>`** no se ejecuta.

Por tanto, cuando el flujo de un algoritmo llega a un bucle **`mientras`**, existen dos posibilidades:

1. Si la condición se evalúa a **falsa**, el bloque de instrucciones no se ejecuta, y el bucle **`mientras`** finaliza sin realizar ninguna iteración.
2. Si la condición se evalúa a **verdadera**, el bloque de instrucciones sí que se ejecuta y, después, se vuelve a evaluar la condición, para decidir –de nuevo– si el bloque de instrucciones se vuelve a ejecutar o no. Y así sucesivamente, hasta que, la condición sea **falsa**.

Cuando el bloque de instrucciones de un bucle se ejecuta, se dice que se ha producido una **iteración**.

El **<bloque_de_instrucciones>** de un bucle **`mientras`** puede ejecutarse cero o más veces (iteraciones). Si el **<bloque_de_instrucciones>** se ejecuta al menos una vez, seguirá ejecutándose repetidamente, mientras que, la condición sea **verdadera**. Pero, hay que tener cuidado de que el bucle no sea infinito.

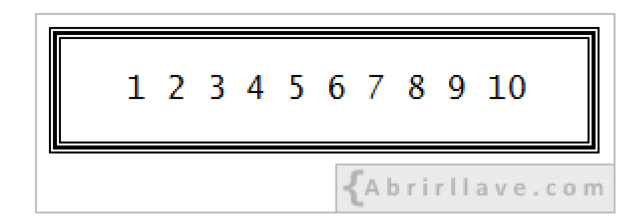
Cuando la condición de un bucle **`mientras`** se evalúa siempre a **verdadera**, se dice que se ha producido un **bucle infinito**, ya que, el algoritmo nunca termina. Un bucle infinito es un error lógico.

Es importante hacer hincapié en el hecho de que, en un bucle **`mientras`**, primero se evalúa la condición y, en el caso de que ésta sea **verdadera**, entonces se ejecuta el bloque de instrucciones. Veremos que, en el bucle **`hacer...mientras`**, el procedimiento es al revés. En él, primero se ejecuta el bloque de instrucciones y, después, se evalúa la condición.

Para que un bucle **`mientras`** no sea infinito, en el bloque de instrucciones debe ocurrir algo para que la condición deje de ser **verdadera**. En la mayoría de los casos, la condición se hace **falsa** al cambiar el valor de una variable.

En resumen, una **instrucción repetitiva `mientras`** permite ejecutar, repetidamente, (cero o más veces) un bloque de instrucciones, mientras que, una determinada condición sea verdadera.

EJEMPLO Para mostrar por pantalla los primeros diez números naturales:



Utilizando un bucle **`mientras`**, se puede escribir el siguiente algoritmo:

```
algoritmo Numeros_del_1_al_10

variables
   entero contador

inicio
   contador ← 1    /* Inicialización del contador */
   mientras ( contador <= 10 )     /* Condición */
      escribir( contador )         /* Salida */
      contador ← contador + 1      /* Incremento */
   fin_mientras
fin
```

El código de este algoritmo es mucho más reducido –más óptimo– que los de los ejemplos del apartado anterior *"10.1. Instrucciones repetitivas"*. Para comprender su funcionamiento, se va a estudiar su traza.

La **traza** de un algoritmo indica la secuencia de acciones (instrucciones) de su ejecución, así como, el valor de las variables del algoritmo después de cada acción (instrucción).

La traza de este algoritmo es:

Secuencia:	Acción (instrucción):	Valor de: Contador
1	`contador ← 1`	1
2	(Comprobar si `contador <= 10`)	1
	La condición es **verdadera**. Inicio de la iteración 1.	
3	`escribir( contador )`	1
4	`contador ← contador + 1`	2
	Fin de la iteración 1.	
5	(Comprobar si `contador <= 10`)	2
	La condición es **verdadera**. Inicio de la iteración 2.	
6	`escribir( contador )`	2
7	`contador ← contador + 1`	3
	Fin de la iteración 2.	

...

n-3	(Comprobar si `contador <= 10`)	10
	La condición es **verdadera**. Inicio de la iteración 10.	
n-2	**`escribir( contador )`**	10
n-1	`contador ← contador + 1`	11
	Fin de la iteración 10.	
n	(Comprobar si `contador <= 10`)	11
	La condición es **falsa**. El bucle finaliza después de 10 iteraciones.	

Explicación de la traza:

- Primeramente, se le asigna el valor 1 a **`contador`** (acción 1).
- En segundo lugar, se evalúa la condición (**`contador <= 10`**) (acción 2) y, puesto que es **verdadera**, se ejecuta el bloque de instrucciones del bucle **`mientras`**.
- Así que, por pantalla se muestra el valor de **`contador`** (acción 3) y, después, se incrementa en **`1`** el valor de la variable **`contador`** (acción 4).
- Una vez terminada la ejecución del bloque de instrucciones, se vuelve a evaluar la condición (**`contador <= 10`**) (acción 5) y, puesto que es **verdadera**, se ejecuta de nuevo el bloque de instrucciones.
- Y así sucesivamente, mientras que, la condición sea **verdadera**, o dicho de otro modo, hasta que, la condición sea **falsa**.
- En este algoritmo, el bloque de instrucciones del bucle **`mientras`** se ejecuta diez veces (iteraciones).

10.2.1. Variable contador

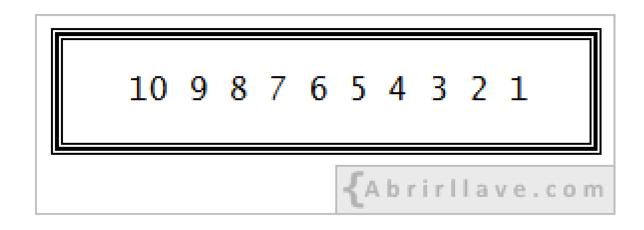
En el algoritmo anterior se ha utilizado un contador. En programación, se llama **contador** a una variable cuyo valor se incrementa o decrementa en un valor fijo (en cada iteración de un bucle).

Un contador suele utilizarse para contar el número de veces que itera un bucle. Pero, a veces, se utiliza para contar, solamente, aquellas iteraciones de un bucle en las que se cumpla una determinada condición.

Además, en este caso, el valor de la variable **`contador`** se ha visualizado en cada iteración.

EJEMPLO Primeros diez números naturales (del 10 al 1).

Se quiere diseñar el algoritmo de un programa que muestre por pantalla los primeros diez números naturales, pero a la inversa, es decir, del 10 al 1:



El algoritmo propuesto es muy similar al anterior, pero, con unos ligeros cambios:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   contador ← 10                    /* Cambio 1 */
   mientras ( contador >= 1 )       /* Cambio 2 */
      escribir( contador )
      contador ← contador - 1       /* Cambio 3 */
   fin_mientras
fin
```

Para que el algoritmo realice la nueva tarea encomendada, ha sido necesario realizar tres cambios en los aspectos más críticos del bucle **mientras**:

- **La inicialización de la variable contador** (cambio 1): necesaria para que la condición pueda evaluarse correctamente cuando el flujo del algoritmo llega al bucle **mientras**.
- **La condición del bucle mientras** (cambio 2): afecta al número de iteraciones que va a efectuar el bucle. También se le conoce como condición de salida del bucle.
- **La instrucción de asignación** (cambio 3): hace variar el valor de la variable **contador** dentro del bloque de instrucciones. De no hacerse correctamente, el bucle podría ser infinito.

EJEMPLO Un pequeño descuido, como por ejemplo, no escribir de forma correcta la condición del bucle, puede producir un bucle infinito:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   contador ← 10                 /* Cambio 1 */
   mientras ( contador <= 10 )   /* Descuido */
      escribir( contador )
      contador ← contador - 1    /* Cambio 3 */
   fin_mientras
fin
```

Por pantalla se mostrará:

```
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 ...
```

{Abrirllave.com

El bucle es infinito. Para que el algoritmo funcionase correctamente, la condición debería ser

```
contador >= 1
```

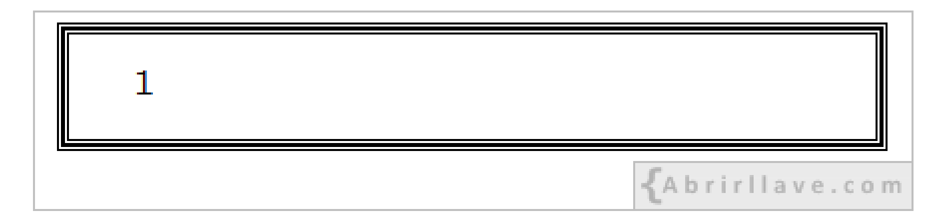
EJEMPLO Otro error muy frecuente es inicializar mal la variable que participa en la condición del bucle:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   contador ← 1                    /* Descuido */
   mientras ( contador >= 1 )      /* Cambio 2 */
      escribir( contador )
      contador ← contador - 1      /* Cambio 3 */
   fin_mientras
fin
```

Por pantalla únicamente se mostrará el número 1:



En este caso, la variable **`contador`** ha sido mal inicializada, y el bucle solamente se ejecuta una vez, ya que, **`contador`** debería haberse inicializado al valor **`10`**.

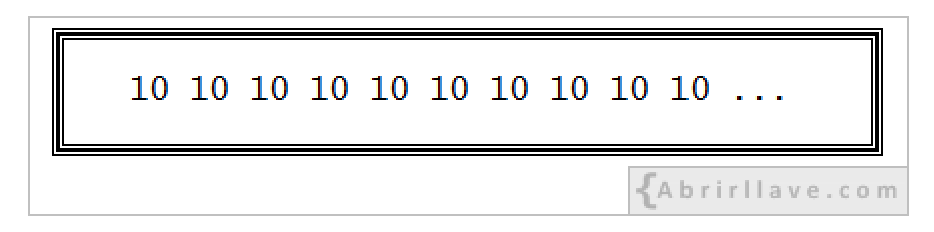
EJEMPLO También es un error muy típico olvidarse de escribir alguna instrucción, como por ejemplo, la instrucción de asignación (**`contador ← contador - 1`**) del bloque de instrucciones del bucle:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   contador ← 10                  /* Cambio 1 */
   mientras ( contador >= 1 )     /* Cambio 2 */
      escribir( contador )
                                  /* Descuido */
   fin_mientras
fin
```

De nuevo, por pantalla, se obtiene la salida de un bucle infinito. En este caso:



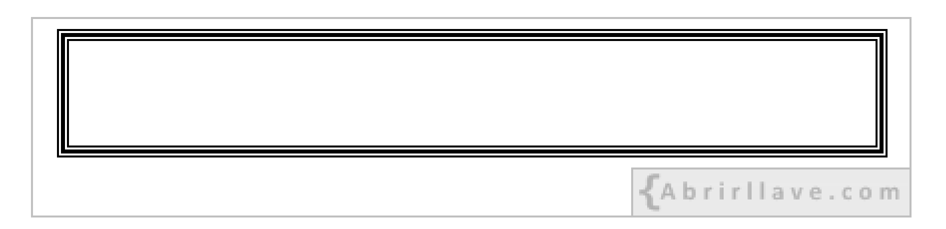
EJEMPLO Como ya se ha dicho, un bucle **mientras** puede iterar cero o más veces. Así, por ejemplo, en el algoritmo siguiente existe un error lógico que provoca que el bucle no itere ninguna vez:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   contador ← 0                   /* Descuido */
   mientras ( contador >= 1 )     /* Cambio 2 */
      escribir( contador )
      contador ← contador - 1     /* Cambio 3 */
   fin_mientras
fin
```

Por pantalla no se mostrará nada:



En este caso, se ha producido un error lógico, ya que, para que el bucle iterase diez veces, se debería haber asignado a la variable **contador** el valor **10**, en vez del **0**.

No obstante, bajo determinadas circunstancias, sí puede tener sentido hacer uso de un bucle **mientras**, el cual pueda no iterar ninguna vez. Por ejemplo, en el siguiente problema.

EJEMPLO Calificación según nota.

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado la nota (dato real) de una asignatura.

2º) En el caso de que la nota sea incorrecta, muestre por pantalla el mensaje:

- "ERROR: Nota incorrecta, debe ser >= 0 y <= 10".

3º) Repita los pasos 1º y 2º, mientras que, la nota introducida sea incorrecta.

4º) Muestre por pantalla:

- "APROBADO", en el caso de que la nota sea mayor o igual que 5.
- "SUSPENDIDO", en el caso de que la nota sea menor que 5.

En pantalla:

```
Introduzca nota (real): 12.4
ERROR: Nota incorrecta, debe ser >= 0 y <= 10
Introduzca nota (real): -3.3
ERROR: Nota incorrecta, debe ser >= 0 y <= 10
Introduzca nota (real): 8.7
APROBADO
```


El algoritmo propuesto es:

```
algoritmo Calificacion_segun_nota

variables
   real nota

inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   /* Si la primera nota introducida por el usuario
      es correcta, el bucle no itera ninguna vez. */

   mientras ( nota < 0 o nota > 10 )
      escribir( "ERROR: Nota incorrecta, debe ser >= 0 y <= 10" )
      escribir( "Introduzca nota (real): " )
      leer( nota )
   fin_mientras

   /* Mientras que el usuario introduzca una nota
      incorrecta, el bucle iterará. Y cuando introduzca
      una nota correcta, el bucle finalizará. */

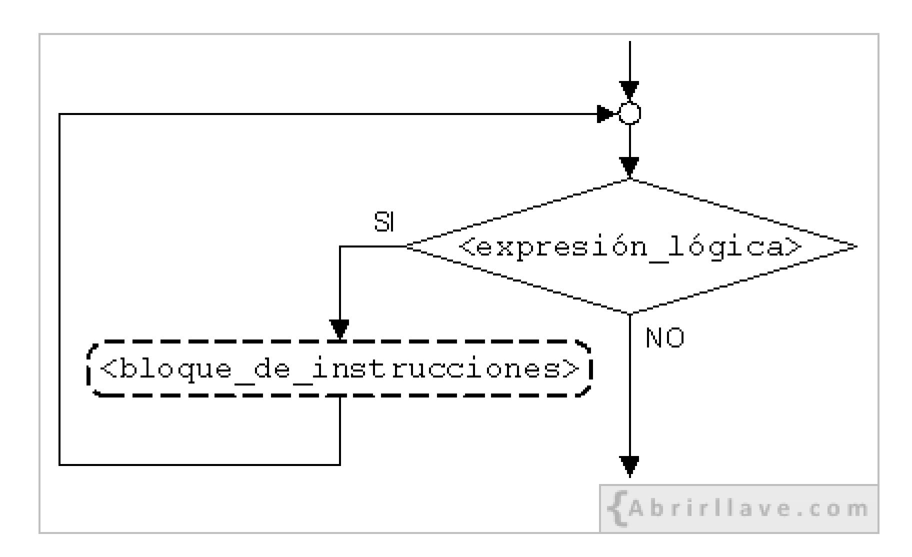
   si ( nota >= 5 )
      escribir( "APROBADO" )
   sino
      escribir( "SUSPENDIDO" )
   fin_si
fin
```

En el algoritmo, el bucle **`mientras`** se ha usado para validar la nota introducida por el usuario. En programación, es muy frecuente usar el bucle **`mientras`** para validar (filtrar) datos. Al bucle que se utiliza para validar uno o más datos, también se le conoce como **filtro**.

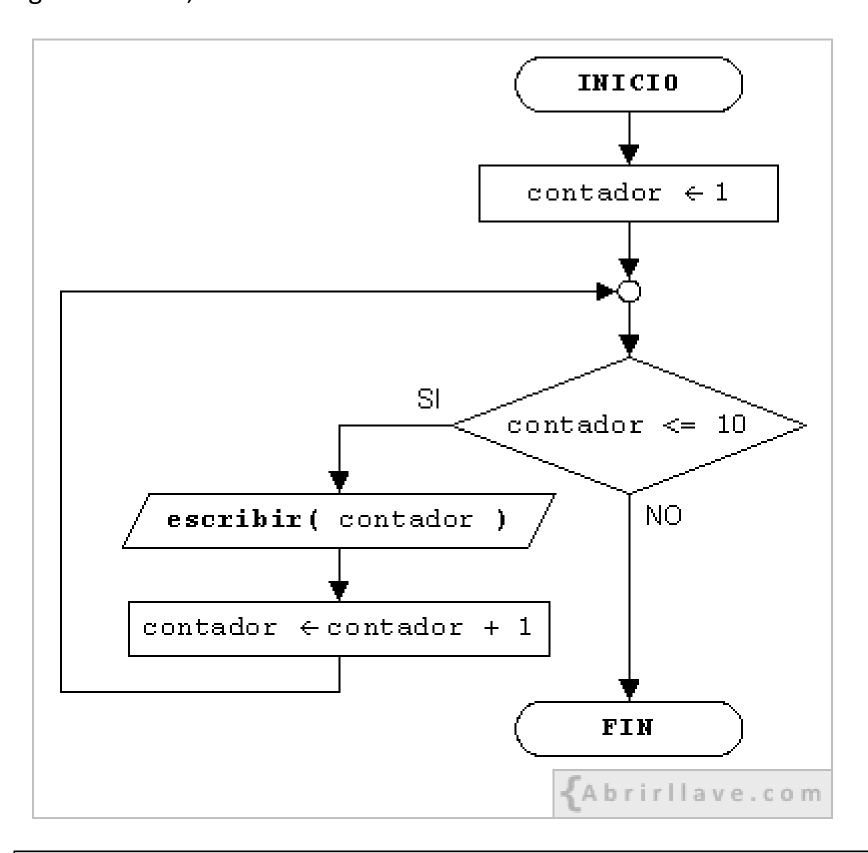
Han aparecido dos nuevas palabras reservadas: **`mientras`** y **`fin_mientras`**. Además, han vuelto a aparecer los símbolos reservados *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**", con un significado distinto.

()	Delimitan la condición en una instrucción repetitiva **`mientras`**.

En un ordinograma, una instrucción repetitiva **`mientras`** se representa de la siguiente manera:



De forma que, por ejemplo, el algoritmo **`Numeros_del_1_al_10`** se puede representar, gráficamente, como se muestra a continuación:



Ejercicios resueltos
• [Ejercicios de la instrucción repetitiva **`mientras`**]()

10.3. Repetitiva `hacer...mientras`

En pseudocódigo, para escribir una instrucción repetitiva **`hacer...mientras`** se utiliza la sintaxis:

```
hacer
   <bloque_de_instrucciones>
mientras( <expresión_lógica> )

/* Ejecutar el <bloque_de_instrucciones> mientras que
   la <expresión_lógica> sea verdadera. */
```

Como se puede apreciar, la instrucción repetitiva **`hacer...mientras`**, también hace uso de una condición.

En un bucle **`hacer...mientras`**, primero se ejecuta el bloque de instrucciones y, después, se evalúa la condición. En el caso de que esta sea **verdadera**, se vuelve a ejecutar el bloque de instrucciones. Y así sucesivamente, hasta que, la condición sea **falsa**.

Por consiguiente, cuando el flujo de un algoritmo llega a un bucle **`hacer...mientras`**, existen dos posibilidades:

1. Se ejecuta el bloque de instrucciones y, después, si la condición se evalúa a **falsa**, el bloque de instrucciones no se vuelve a ejecutar, de manera que, el bucle **`hacer...mientras`** finaliza, habiendo realizado una sola iteración.
2. Se ejecuta el bloque de instrucciones y, a continuación, si la condición se evalúa a **verdadera**, el bloque de instrucciones se vuelve a ejecutar. Y así sucesivamente, hasta que la condición sea **falsa**.

El **<bloque_de_instrucciones>** de un bucle **`hacer...mientras`** puede ejecutarse una o más veces (iteraciones). También hay que prevenir que el bucle no sea infinito.

En resumen, una **instrucción repetitiva `hacer...mientras`** permite ejecutar repetidamente (una o más veces) un bloque de instrucciones, mientras que, una determinada condición sea **verdadera**.

EJEMPLO Para mostrar por pantalla los primeros diez números naturales (del 1 al 10), utilizando un bucle **`hacer...mientras`** se puede escribir el siguiente código:

```
algoritmo Numeros_del_1_al_10

variables
   entero contador

inicio
   contador ← 1   /* Inicialización del contador */
   hacer
      escribir( contador )         /* Salida */
      contador ← contador + 1      /* Incremento */
   mientras( contador <= 10 )      /* Condición */
fin
```

La traza de este algoritmo es:

Secuencia:	Acción (instrucción):	Valor de: contador
1	`contador ← 1`	1
	Inicio de la iteración 1.	
2	`escribir( contador )`	[1]
3	`contador ← contador + 1`	2
	Fin de la iteración 1.	
4	(Comprobar si `contador <= 10`)	2
	La condición es **verdadera**. Inicio de la iteración 2.	
5	`escribir( contador )`	[2]
6	`contador ← contador + 1`	3
	Fin de la iteración 2.	
...		
n-3	(Comprobar si `contador <= 10`)	10
	La condición es **verdadera**. Inicio de la iteración 10.	
n-2	`escribir( contador )`	[10]
n-1	`contador ← contador + 1`	11
	Fin de la iteración 10.	
n	(Comprobar si `contador <= 10`)	11
	La condición es **falsa**. El bucle finaliza después de 10 iteraciones.	

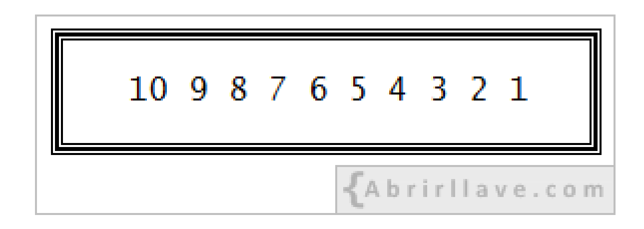
Explicación de la traza:

- En primer lugar, se le asigna el valor **1** a **contador** (acción 1).
- A continuación, se ejecuta el bloque de instrucciones del bucle **hacer... mientras**, mostrándose por pantalla el valor de **contador** (acción 2) y, después, se incrementa en **1** el valor de la variable **contador** (acción 3).

- Una vez ejecutado el bloque de instrucciones, se evalúa la condición de salida del bucle (**`contador <= 10`**) (acción 4) y, puesto que, es **verdadera**, se ejecuta –de nuevo– el bloque de instrucciones.
- Y así sucesivamente, mientras que, la condición sea **verdadera**, o dicho de otro modo, hasta que, la condición sea **falsa**.
- En este algoritmo, el bloque de instrucciones del bucle se ejecuta diez veces (iteraciones).

EJEMPLO Primeros diez números naturales (del 10 al 1).

Utilizando un bucle **`hacer...mientras`**, se quiere diseñar el algoritmo de un programa que muestre por pantalla los primeros diez números naturales, pero a la inversa, es decir, del 10 al 1:



Para ello, se propone el siguiente algoritmo, muy similar al anterior, pero con unos ligeros cambios:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

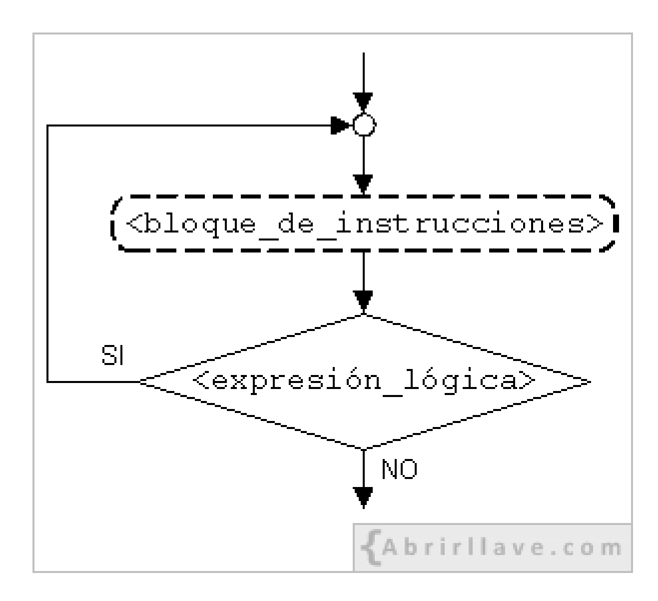
inicio
   contador ← 10                /* Cambio 1 */
   hacer
      escribir( contador )
      contador ← contador - 1    /* Cambio 2 */
   mientras( contador >= 1 )     /* Cambio 3 */
fin
```

Para que el algoritmo realice la nueva tarea, ha sido necesario realizar tres cambios en los aspectos más críticos del bucle **`hacer...mientras`**:

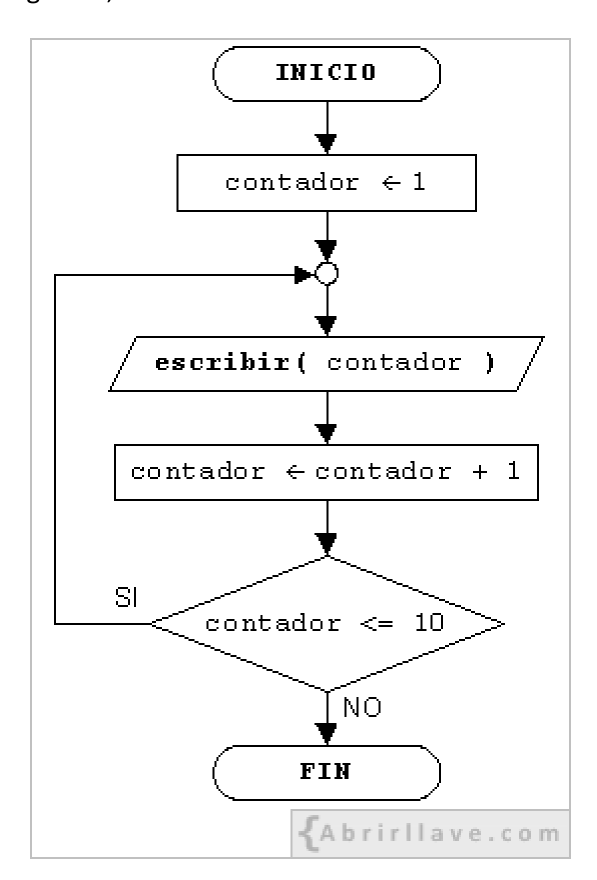
- <u>La inicialización de la variable **`contador`**</u> (cambio 1): necesaria para que, en la primera iteración, el bloque de instrucciones del bucle pueda ejecutarse correctamente, tanto la instrucción de salida como la de asignación.
- <u>La instrucción de asignación</u> (cambio 2): hace variar el valor de la variable **`contador`** dentro del bloque de instrucciones. De no hacerse correctamente, el bucle podría ser infinito.
- <u>La condición del bucle **`hacer...mientras`**</u> (cambio 3): afecta al número de iteraciones que va a efectuar el bucle. También se le conoce como condición de salida del bucle.

Al igual que ocurre con la instrucción repetitiva **mientras**, cualquier pequeño descuido o error al escribir el código del algoritmo, puede dar lugar a que la instrucción repetitiva **hacer...mientras** no funcione correctamente.

En un ordinograma, una instrucción repetitiva **hacer...mientras** se representa de la siguiente manera:



Así, por ejemplo, el algoritmo **Numeros_del_1_al_10** se puede representar, de manera gráfica, como se muestra a continuación:



Como ya se ha dicho, el bucle **hacer...mientras** puede iterar una o más veces. Por tanto, cuando un bloque de instrucciones debe iterar al menos una vez, generalmente, es mejor utilizar un bucle **hacer...mientras** que un bucle **mientras**, como por ejemplo, en el siguiente problema.

EJEMPLO Suma de números introducidos por el usuario.

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado un número (dato entero).

2º) Pregunte al usuario si desea introducir otro o no.

3º) Repita los pasos 1º y 2º, mientras que, el usuario no responda 'n' de (no).

4º) Muestre por pantalla la suma de los números introducidos por el usuario.

En pantalla:

```
Introduzca un número entero: 7
¿Desea introducir otro (s/n)?: s
Introduzca un número entero: 16
¿Desea introducir otro (s/n)?: s
Introduzca un número entero: -3
¿Desea introducir otro (s/n)?: n
La suma de los números introducidos es: 20
```

```
Introduzca un número entero: 11
¿Desea introducir otro (s/n)?: n
La suma de los números introducidos es: 11
```


Solución:

```
algoritmo Suma_de_numeros_introducidos_por_el_usuario

variables
   caracter seguir
     entero acumulador, numero

inicio

   /* En acumulador se va a guardar la suma
      de los números introducidos por el usuario. */

   acumulador ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( numero )
      acumulador ← acumulador + numero
      escribir( "¿Desea introducir otro número (s/n)?: " )
      leer( seguir )
   mientras( seguir <> 'n' )

   /* Mientras que el usuario desee introducir más números,
      el bucle iterará. */

   escribir( "La suma de los números introducidos es: ",
             acumulador )
fin
```

En la solución propuesta, cuando el programa pregunta al usuario si desea introducir otro número, el bucle iterará, mientras que, seguir sea distinto de **'n'**. De manera que, cualquier otro carácter que no sea **'n'** provocará que el bucle itere de nuevo. En pantalla:

```
Introduzca un número entero: 7
¿Desea introducir otro (s/n)?: j
Introduzca un número entero: 16
¿Desea introducir otro (s/n)?: t
Introduzca un número entero: -3
¿Desea introducir otro (s/n)?: n
La suma de los números introducidos es: 20
```


10.3.1. Variable acumulador

En el algoritmo anterior se ha utilizado un acumulador. En programación, se llama **acumulador** a una variable cuyo valor se incrementa o decrementa en un valor que no tiene por qué ser fijo (en cada iteración de un bucle).

Un acumulador suele utilizarse para acumular resultados producidos en las iteraciones de un bucle.

10.3.2. Diferencias entre un bucle `mientras` y `hacer...mientras`

Para saber cuándo hacer uso de un bucle u otro, es muy importante conocer bien las diferencias más significativas existentes entre ambos bucles.

Diferencias entre un bucle* `mientras` *y un bucle* `hacer...mientras`*:

mientras	**hacer...mientras**
Pasos a realizar:	
1. Se evalúa la condición.	1. Se ejecuta el bloque de instrucciones.
2. Se ejecuta el bloque de instrucciones.	2. Se evalúa la condición.
Número de iteraciones:	
Cero o más veces (0-n).	Una o más veces (1-n).

{Abrirllave.com

Ha aparecido una nueva palabra reservada: **`hacer`**.

Han vuelto a aparecer los símbolos reservados *abrir paréntesis* "**(**" y *cerrar paréntesis* "**)**", con un significado distinto.

()	Delimitan la condición en una instrucción repetitiva **`hacer...mientras`**.

Ejercicios resueltos
• Ejercicios de la instrucción repetitiva **`hacer...mientras`**

10.4. Repetitiva `para`

En pseudocódigo, para escribir una instrucción repetitiva **`para`** se utiliza la sintaxis:

```
para <variable> ← <valor_inicial> hasta <valor_final>
[ incremento <valor_incremento> ] hacer
   <bloque_de_instrucciones>
fin_para

/* A la <variable> se le asigna el <valor_inicial> y,
   se le suma, sucesivamente, el <valor_incremento>,
   hasta superar el <valor_final>. */

/* Por omisión, el <valor_incremento> es 1. */
```

En una instrucción repetitiva **para**, siempre se utiliza una **<variable>** a la que se debe asignar un **<valor_inicial>**. En cada iteración del bucle, al valor de la **<variable>** se le suma el **<valor_incremento>** y, cuando la **<variable>** supera el **<valor_final>**, el bucle finaliza.

En consecuencia, cuando el flujo de un algoritmo llega a un bucle **para**, en primer lugar, se asigna el **<valor_inicial>** a la **<variable>** y, a partir de ese instante, existen dos posibilidades:

1. Si el valor de la **<variable>** es mayor que el **<valor_final>**, entonces no se ejecuta el bloque de instrucciones, y el bucle **para** finaliza sin realizar ninguna iteración.
2. Si el valor de la **<variable>** es menor o igual que el **<valor_final>**, entonces, se ejecuta el bloque de instrucciones y, después, se le suma el **<valor_incremento>** a la **<variable>**, volviéndose, de nuevo, a comparar el valor de la **<variable>** con el **<valor_final>**. Y así sucesivamente, hasta que, el valor de la **<variable>** sea mayor que el **<valor_final>**.

En resumen, una **instrucción repetitiva para** permite ejecutar, repetidamente, un bloque de instrucciones, en base a un valor inicial y a un valor final.

El bucle **para** es ideal usarlo cuando, de antemano, ya se sabe el número de veces (iteraciones) que tiene que ejecutarse un determinado bloque de instrucciones.

El bucle **para** es una variante del bucle **mientras** y, al igual que éste, puede iterar cero o más veces. Sin embargo, el bucle **para** solamente se suele usar cuando se conoce el número exacto de veces que tiene que iterar el bucle. Este es el caso cuando se quiere mostrar por pantalla los primeros diez números naturales (del 1 al 10), donde se sabe de antemano que el bucle tiene que iterar, exactamente, diez veces.

EJEMPLO Por tanto, el algoritmo **Numeros_del_1_al_10** se puede resolver con una instrucción repetitiva **para** de la siguiente forma:

```
algoritmo Numeros_del_1_al_10

variables
   entero contador

inicio
   para contador ← 1 hasta 10 incremento 1 hacer
      escribir( contador )
   fin_para
fin
```

EJEMPLO Cuando el incremento es **1**, se puede omitir la palabra reservada **incremento**, y su valor:

```
algoritmo Numeros_del_1_al_10
variables
   entero contador

inicio
   /* Al no aparecer el valor del incremento,
      se entiende que es 1. */

   para contador ← 1 hasta 10 hacer
      escribir( contador )
   fin_para
fin
```

La traza de ambos algoritmos es la misma:

Secuencia:	Acción (instrucción):	Valor de: contador
1	contador ← 1	1
2	(Comprobar si contador es menor o igual que 10)	1
	contador sí es menor o igual que 10. Inicio de la iteración 1.	
3	escribir(contador)	1
	Fin de la iteración 1.	
4	(Sumar a contador el valor 1)	2
5	(Comprobar si contador es menor o igual que 10)	2
	contador sí es menor o igual que 10. Inicio de la iteración 2.	
6	escribir(contador)	2
	Fin de la iteración 2.	

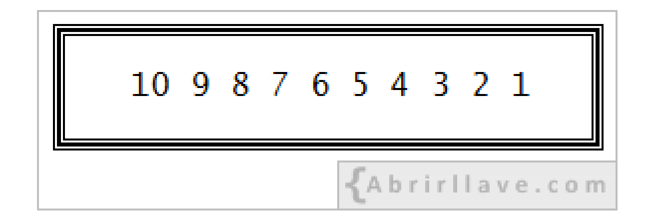
...		
n-3	(Comprobar si `contador` es menor o igual que 10)	10
	`contador` sí es menor o igual que 10. Inicio de la iteración 10.	
n-2	**`escribir( contador )`**	10
	Fin de la iteración 10.	
n-1	(Sumar a `contador` el valor 1)	11
n	(Comprobar si `contador` es menor o igual que 10)	11
	`contador` no es menor o igual que 10. El bucle finaliza después de 10 iteraciones.	

Explicación de la traza:

- Primeramente, se le asigna el valor 1 a **`contador`** (acción 1).
- A continuación, se comprueba si **`contador`** es menor o igual que 10 (acción 2) y, puesto que esto es **verdadero**, se ejecuta el bloque de instrucciones del bucle **`para`** (una sola instrucción en este caso).
- Así pues, se muestra por pantalla el valor de **`contador`** (acción 3).
- Después, se incrementa en **`1`** el valor de la variable **`contador`** (acción 4).
- Posteriormente, se vuelve a comprobar si **`contador`** es menor o igual que `10` (acción 5).
- Y así sucesivamente, mientras que, el valor de **`contador`** sea menor o igual que `10`, o dicho de otro modo, hasta que, el valor de contador sea mayor que `10`.
- En este algoritmo, el bloque de instrucciones del bucle **`para`** se ejecuta diez veces (iteraciones).

EJEMPLO Primeros diez números naturales (del 10 al 1).

Utilizando un bucle **`para`**, se quiere diseñar el algoritmo de un programa que muestre por pantalla los primeros diez números naturales, pero a la inversa, es decir, del 10 al 1:



Para ello, debe utilizarse un incremento negativo:

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   para contador ← 10 hasta 1 incremento -1 hacer
      escribir( contador )
   fin_para
fin
```

En este algoritmo, el **<valor_inicial>** y el **<valor_final>** son el **10** y el **1**, respectivamente, al revés que en el algoritmo anterior. De manera que, el bucle iterará, hasta que, el valor de contador sea menor que **1**, o dicho de otro modo, mientras que, el valor de contador sea mayor o igual que **1**.

EJEMPLO Por otra parte, también es posible omitir la palabra reservada **incremento** y su valor, entendiéndose, en ese caso, que es **-1**, ya que, el **<valor_inicial>** es mayor que el **<valor_final>** y, por tanto, solamente es razonable un incremento negativo.

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   /* Al no aparecer el valor del incremento,
      se entiende que es -1. */

   para contador ← 10 hasta 1 hacer
      escribir( contador )
   fin_para
fin
```

Para los casos en que el incremento es negativo, también se puede utilizar la sintaxis:

```
para <variable> ← <valor_inicial> hasta <valor_final>
[ decremento <valor_decremento> ] hacer
   <bloque_de_instrucciones>
fin_para

/* En vez de "incremento", se utiliza "decremento". */
```

EJEMPLO Primeros diez números naturales (del 10 al 1) utilizando un bucle **para** y la palabra reservada **decremento**.

```
algoritmo Numeros_del_10_al_1

variables
   entero contador

inicio
   para contador ← 10 hasta 1 decremento 1 hacer
      escribir( contador )
   fin_para
fin
```

Por consiguiente, la sintaxis completa de una instrucción repetitiva **para** es:

```
para <variable> ← <valor_inicial> hasta <valor_final>
[ incremento <valor_incremento> |
  decremento <valor_decremento> ] hacer
   <bloque_de_instrucciones>
fin_para
```

El carácter tubería (**|**) se utiliza para indicar que, o bien se escribe:

```
incremento <valor_incremento>
```

O bien se escribe:

```
decremento <valor_incremento>
```

Pero, no ambos.

Es muy poco probable que, por equivocación, un programador diseñe un bucle infinito con una instrucción repetitiva **para**. Aun así, siempre cabe la posibilidad de que esto suceda.

EJEMPLO En el siguiente algoritmo hay un bucle infinito.

```
algoritmo Numeros_del_1_al_10

variables
   entero contador

inicio
   para contador ← 1 hasta 10 incremento -1 hacer
      escribir( contador )
   fin_para
fin
```

En pantalla, se mostrará:

```
1 0 -1 -2 -3 -4 ...
```


Dentro del bloque de instrucciones de un bucle **para**, no es recomendable cambiar el valor de la variable utilizada para controlar el número de iteraciones. Hacer esto, puede llevar a que el bucle no funcione de la manera esperada, produciendo errores inesperados.

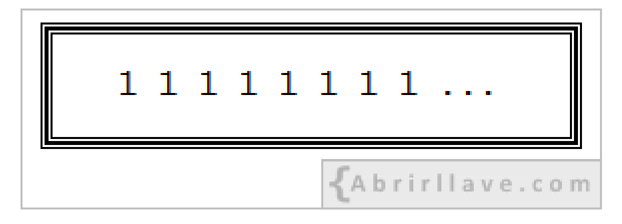
EJEMPLO En el siguiente algoritmo, al cambiar el valor de **contador** dentro del bloque de instrucciones, se ha producido un bucle infinito.

```
algoritmo Numeros_del_1_al_10

variables
   entero contador

inicio
   para contador ← 1 hasta 10 hacer
      escribir( contador )
      contador ← contador - 1
   fin_para
fin
```

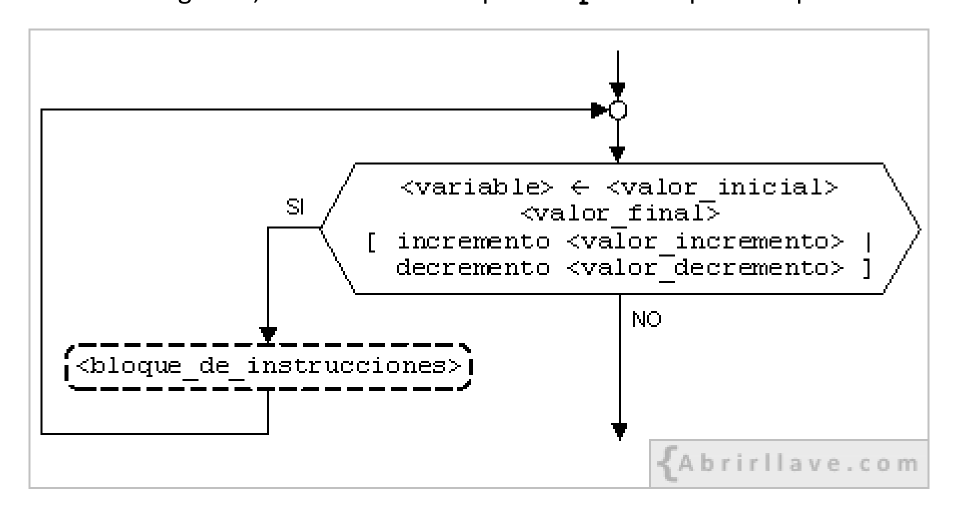
En pantalla se mostrará:



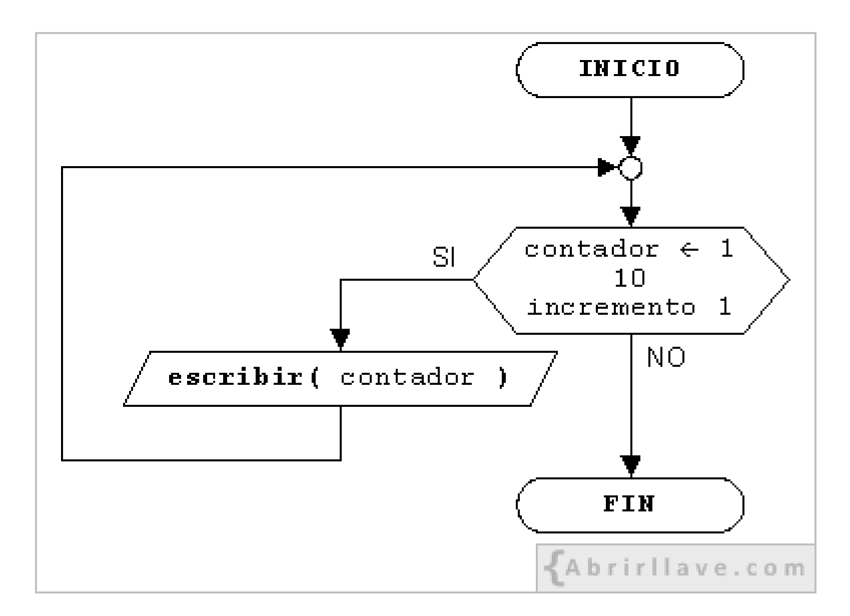
Han aparecido cinco nuevas palabras reservadas: **para**, **hasta**, **incremento**, **decremento** y **fin_para**.

También ha vuelto a aparecer la palabra reservada **hacer**.

En un ordinograma, una instrucción repetitiva **para** se puede representar del siguiente modo:



De forma que, por ejemplo, el algoritmo **`Numeros_del_1_al_10`** se puede representar, gráficamente, de la siguiente forma:



10.5. Cuándo usar un bucle u otro

A la hora de elegir un bucle u otro, debemos hacernos la siguiente pregunta:

- **¿Se conoce, de antemano, el número de veces (iteraciones) que tiene que ejecutarse un determinado bloque de instrucciones?**

Si la respuesta es afirmativa, habitualmente se usa un bucle **`para`**. En caso contrario, nos plantearemos la siguiente pregunta:

- **¿El bloque de instrucciones debe ejecutarse al menos una vez?**

En este caso, si la respuesta es afirmativa, generalmente haremos uso de un bucle **`hacer...mientras`**, y si la respuesta es negativa, usaremos un bucle **`mientras`**.

Ejercicios resueltos
• Ejercicios de la instrucción repetitiva **`para`**

10.6. Anidamiento de alternativas y repetitivas

Al igual que las instrucciones alternativas, las instrucciones repetitivas también se pueden anidar, permitiendo las siguientes combinaciones de anidamiento:

- **`mientras`** en **`mientras`**.
- **`mientras`** en **`hacer...mientras`**.
- **`mientras`** en **`para`**.

- **hacer...mientras** en **hacer...mientras**.
- **hacer...mientras** en **para**.
- **hacer...mientras** en **mientras**.
- **para** en **para**.
- **para** en **mientras**.
- **para** en **hacer...mientras**.

De ellas, vamos a estudiar, como ejemplo, las combinaciones:

- **para** en **para**.
- **para** en **hacer...mientras**.

Para comprender la utilidad del anidamiento de bucles, estúdiese el problema del ejemplo siguiente.

EJEMPLO Tablas de multiplicar del 1, 2 y 3.

Se quiere diseñar el algoritmo de un programa que muestre por pantalla las tablas de multiplicar del 1, 2 y 3:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```


Una solución al problema es:

```
algoritmo Tablas_de_multiplicar_del_1_2_y_3

variables
   entero i

inicio
   /* Tabla de multiplicar del 1 */
   para i ← 1 hasta 10 hacer
      escribir( "1 * ", i, " = ", i )
   fin_para

   /* Tabla de multiplicar del 2 */
   para i ← 1 hasta 10 hacer
      escribir( "2 * ", i, " = ", i * 2 )
   fin_para

   /* Tabla de multiplicar del 3 */
   para i ← 1 hasta 10 hacer
      escribir( "3 * ", i, " = ", i * 3 )
   fin_para
fin
```

En el algoritmo, para mostrar las tres tablas de multiplicar, se han escrito tres bucles **para**. Sin embargo, haciendo uso del anidamiento, se puede reducir el código del algoritmo.

10.6.1. Bucle para en para

En pseudocódigo, para anidar una instrucción repetitiva **para** en otra, se utiliza la sintaxis:

```
para <variable_1> ← <valor_inicial_1> hasta
<valor_final_1> [ incremento <valor_incremento_1> |
                  decremento <valor_decremento_1> ] hacer

   /* Inicio del anidamiento */

   para <variable_2> ← <valor_inicial_2> hasta
   <valor_final_2> [ incremento <valor_incremento_2> |
                     decremento <valor_decremento_2> ] hacer
      <bloque_de_instrucciones>
   fin_para

   /* Fin del anidamiento */

fin_para
```

Cuando un bucle se anida dentro de otro, el número de iteraciones del **<bloque_de_instrucciones>** se multiplica. En el algoritmo **Tablas_de_multiplicar_del_1_2_y_3**, se han escrito tres bucles **para**, iterando cada uno de ellos 10 veces, haciendo un total de 30 iteraciones (3*10). Pues bien, anidando un bucle dentro de otro, se puede realizar el

mismo número de iteraciones, reduciendo el código del algoritmo (solamente se van a necesitar dos bucles, en vez de tres).

Si se pretendiese mostrar por pantalla, por ejemplo, las tablas de multiplicar del 1 al 10, todavía sería mayor la reducción del código del algoritmo. En ese caso, en vez de escribir diez bucles, iterando cada uno de ellos otras 10 veces, haciendo un total de 100 iteraciones (10*10), sería mejor anidar un bucle dentro de otro, reduciéndose considerablemente el código del algoritmo (solamente se necesitarían dos bucles, en vez de diez).

EJEMPLO Así pues, anidando un bucle **para** en otro, se pueden mostrar por pantalla las tablas de multiplicar del 1, 2 y 3:

```
algoritmo Tablas_de_multiplicar_del_1_2_y_3

variables
   entero i, j, r

inicio
   para i ← 1 hasta 3 hacer  /* Bucle 1 */

      /* Inicio del anidamiento */
      para j ← 1 hasta 10 hacer  /* Bucle 2(anidado) */
         r ← i * j
         escribir( i, " * ", j, " = ", r )
      fin_para
      /* Fin del anidamiento */

   fin_para
fin
```

10.6.2. Bucle para en hacer...mientras

En pseudocódigo, para anidar un bucle **para** en un bucle **hacer...mientras**, se utiliza la sintaxis:

```
hacer

   /* Inicio del anidamiento */
   para <variable> ← <valor_inicial> hasta <valor_final>
   [ incremento <valor_incremento> |
     decremento <valor_decremento> ] hacer
      <bloque_de_instrucciones>
   fin_para
   /* Fin del anidamiento */

mientras ( <expresión_lógica> )
```

EJEMPLO Tabla de multiplicar de un número.

Se quiere diseñar el algoritmo de un programa que muestre por pantalla la tabla de multiplicar de un número entero introducido por el usuario. El proceso debe repetirse mientras que el usuario lo desee:

```
Introduzca un número entero: 7

La tabla de multiplicar del 7 es:

7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70

¿Desea ver otra tabla (s/n)?: s

Introduzca número entero: -12

La tabla de multiplicar del -12 es:

-12 * 1 = -12
-12 * 2 = -24
-12 * 3 = -36
-12 * 4 = -48
-12 * 5 = -60
-12 * 6 = -72
-12 * 7 = -84
-12 * 8 = -96
-12 * 9 = -108
-12 * 10 = -120

¿Desea ver otra tabla (s/n)?: n
```


Algoritmo propuesto:

```
algoritmo Tabla_de_multiplicar_de_un_numero

variables
   caracter seguir
     entero i, numero

inicio
   hacer
      escribir( "Introduzca un número entero: " )
      leer( numero )

      escribir( "La tabla de multiplicar del ", numero , " es: " )

      /* Inicio del anidamiento */
      para i ← 1 hasta 10 hacer
         escribir( numero, " * ", i, " = ", i * numero )
      fin_para
      /* Fin del anidamiento */

      escribir( "¿Desea ver otra tabla (s/n)?: " )
      leer( seguir )
   mientras ( seguir <> 'n' )
fin
```

Las instrucciones alternativas y repetitivas también se pueden anidar entre sí, permitiendo realizar 18 combinaciones más de anidamiento:

- **mientras** en doble.
- **mientras** en simple.
- **mientras** en múltiple.
- **hacer...mientras** en doble.
- **hacer...mientras** en simple.
- **hacer...mientras** en múltiple.
- **para** en doble.
- **para** en simple.
- **para** en múltiple.
- Doble en **mientras**.
- Doble en **hacer...mientras**.
- Doble en **para**.
- Simple en **mientras**.
- Simple en **hacer...mientras**.
- Simple en **para**.
- Múltiple en **mientras**.
- Múltiple en **hacer...mientras**.
- Múltiple en **para**.

De ellas, vamos a estudiar, como ejemplo, las combinaciones:

- Simple en **para**.
- Múltiple en **hacer...mientras**.

10.6.3. Alternativa simple en bucle `para`

En pseudocódigo, para anidar una alternativa simple en un bucle **`para`**, se utiliza la sintaxis:

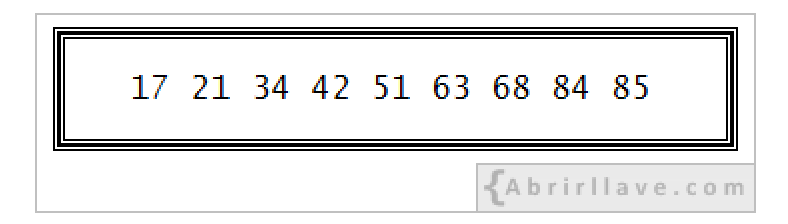
```
para <variable> ← <valor_inicial> hasta <valor_final>
[ incremento <valor_incremento> |
  decremento <valor_decremento> ] hacer

   /* Inicio del anidamiento */
   si ( <expresión_lógica> )
      <bloque_de_instrucciones>
   fin_si
   /* Fin del anidamiento */

fin_para
```

EJEMPLO Números enteros divisibles entre 17 ó 21.

Se quiere diseñar el algoritmo de un programa que muestre por pantalla todos los números enteros del 1 al 100 (ambos inclusive) que sean divisibles entre 17 ó 21:



Anidando una alternativa simple en un bucle **`para`**, el problema se puede resolver con el algoritmo:

```
algoritmo Numeros_enteros_divisibles_entre_17_o_21

variables
   entero numero

inicio
   para numero ← 1 hasta 100 hacer

      /* Inicio del anidamiento */
      si ( numero mod 17 = 0 o numero mod 21 = 0 )
         escribir( numero )
      fin_si
      /* Fin del anidamiento */

   fin_para
fin
```

10.6.4. Alternativa múltiple en bucle `hacer...mientras`

Anidar una alternativa múltiple en un bucle **`hacer...mientras`**, es la solución idónea cuando se quiere realizar un menú. Un **menú** ofrece al usuario la posibilidad de elegir qué acción(es) realizar de entre una lista de opciones.

EJEMPLO Menú de opciones.

Se quiere diseñar el algoritmo de un programa que:

1º) Muestre un menú con 4 opciones:

- 1. Calcular el doble de un número entero.
- 2. Calcular la mitad de un número entero.
- 3. Calcular el cuadrado de un número entero.
- 4. Salir.

2º) Pida por teclado la opción deseada (dato entero).

3º) Ejecute la opción del menú seleccionada.

4º) Repita los pasos 1º, 2º y 3º, mientras que, el usuario no seleccione la opción 4 (Salir) del menú.

En pantalla:

```
1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 3

Introduzca un número entero: 16

El cuadrado de 16 es 256


1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 1

Introduzca un número entero: 19

El doble de 19 es 38


1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 4
```

El algoritmo propuesto es:

```
algoritmo Menu_de_opciones

variables
   entero numero, opcion

inicio
  hacer
     escribir( "1. Calcular el doble de un número entero." )
     escribir( "2. Calcular la mitad de un número entero." )
     escribir( "3. Calcular el cuadrado de un número entero." )
     escribir( "4. Salir." )
     escribir( "Introduzca opción: " )

     leer( opcion )

     /* Inicio del anidamiento */

     segun_sea ( opcion )
        1 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "El doble de ", numero, " es ",
                      numero * 2 )

        2 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "La mitad de ", numero, " es ",
                      numero / 2 )

        3 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "El cuadrado de ", numero, " es ",
                      numero * numero )
     fin_segun_sea

     /* Fin del anidamiento */

  mientras ( opcion <> 4 )
fin
```

En la solución propuesta, el bucle iterará, mientras que, **opcion** sea distinto del valor **4**.

Normalmente, en un menú, la opción de salir –opción 4 en este caso– no se debe contemplar en la alternativa múltiple, es decir, si el usuario introduce un 4, no se debe hacer nada.

Pero, ¿qué ocurre si el usuario teclea un número mayor que 4 ó menor que 1?

En pantalla:

```
1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 5

1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 0

1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 2

Introduzca un número entero: 17

La mitad de 17 es 8.5

1. Calcular el doble de un número entero.
2. Calcular la mitad de un número entero.
3. Calcular el cuadrado de un número entero.
4. Salir.

Introduzca opción (1-4): 4
```


Al introducir un número menor que 1 ó mayor que 4, se muestra de nuevo el menú. Para evitar que ocurra esto, es conveniente utilizar un filtro al leer la opción que introduce el usuario.

Dicho filtro puede ser el siguiente:

```
/* Filtramos la opción elegida por el usuario */
hacer
   escribir( "Introduzca opción: " )
   leer( opcion )
mientras( opcion < 1 o opcion > 4 )
/* La opción solamente puede ser 1, 2, 3 ó 4 */
```

De modo que el código del algoritmo sería:

```
algoritmo Menu_de_opciones

variables
   entero numero, opcion

inicio
  hacer
     escribir( "1. Calcular el doble de un número entero." )
     escribir( "2. Calcular la mitad de un número entero." )
     escribir( "3. Calcular el cuadrado de un número entero." )
     escribir( "4. Salir." )

     /* Filtramos la opción elegida por el usuario */

     hacer
        escribir( "Introduzca opción: " )
        leer( opcion )
     mientras( opcion < 1 o opcion > 4 )

     /* La opción solamente puede ser 1, 2, 3 ó 4 */

     /* Inicio del anidamiento */

     segun_sea ( opcion )
        1 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "El doble de ", numero, " es ",
                      numero * 2 )

        2 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "La mitad de ", numero, " es ",
                      numero / 2 )

        3 : escribir( "Introduzca un número entero:" )
            leer( numero )
            escribir( "El cuadrado de ", numero, " es ",
                      numero * numero )
     fin_segun_sea

     /* Fin del anidamiento */

  mientras ( opcion <> 4 )
fin
```

La variable **opcion**, también puede ser un dato de tipo carácter, en vez de tipo entero:

```
algoritmo Menu_de_opciones

variables
   caracter opcion
     entero numero

inicio
  hacer
     escribir( "1. Calcular el doble de un número entero." )
     escribir( "2. Calcular la mitad de un número entero." )
     escribir( "3. Calcular el cuadrado de un número entero." )
     escribir( "4. Salir." )

     /* Filtramos la opción elegida por el usuario */

     hacer
        escribir( "Introduzca opción: " )
        leer( opcion )
     mientras( opcion < 1 o opcion > 4 )

     /* La opción solamente puede ser 1, 2, 3 ó 4 */

     /* Inicio del anidamiento */

     segun_sea ( opcion )
        '1' : escribir( "Introduzca un número entero:" )
              leer( numero )
              escribir( "El doble de ", numero, " es ",
                        numero * 2 )

        '2' : escribir( "Introduzca un número entero:" )
              leer( numero )
              escribir( "La mitad de ", numero, " es ",
                        numero / 2 )

        '3' : escribir( "Introduzca un número entero:" )
              leer( numero )
              escribir( "El cuadrado de ", numero, " es ",
                        numero * numero )
     fin_segun_sea

     /* Fin del anidamiento */

  mientras ( opcion <> 4 )
fin
```

Ejercicios resueltos

- Ejercicios de instrucciones repetitivas

Capítulo 11

Instrucciones de control de salto

Las instrucciones de control de salto (en especial la instrucción **`ir_a`**) son un lastre de los orígenes de la programación. De hecho, en programación estructurada, todos los programas se pueden escribir utilizando tres tipos de estructuras de control:

- Secuencial.
- De selección (alternativas).
- De iteración (repetitivas).

Así pues, todos los programas que utilizan instrucciones de salto, pueden ser reescritos sin hacer uso de ellas y, aunque muchos lenguajes de programación permiten codificar las instrucciones de salto, el hacer uso de ellas se considera una práctica de programación pobre y nefasta; tal es así, que, ni siquiera las vamos a representar mediante ordinogramas en este tutorial.

Resumiendo, en este capítulo se van a estudiar las instrucciones de control de salto, las ventajas de no hacer uso de ellas y cómo se pueden reescribir los algoritmos que las usan.

11.1. Instrucciones de salto

Las **instrucciones de control de salto** permiten realizar saltos en el flujo de control de un programa, es decir, permiten transferir el control del programa, alterando bruscamente el flujo de control del mismo. Existen cuatro tipos de instrucciones de salto:

- **`interrumpir`** (romper, salir, terminar...)
- **`continuar`**
- **`ir_a`**
- **`volver`**

Cuando en un programa se utiliza una instrucción de salto, la secuencia normal de su ejecución se rompe, transfiriéndose el control del programa a otro lugar dentro del mismo.

11.2. interrumpir

En pseudocódigo, para escribir una **instrucción de salto `interrumpir`** se utiliza la sintaxis:

```
interrumpir
```

En pseudocódigo, la instrucción de salto **`interrumpir`** siempre se usa para interrumpir (romper) la ejecución normal de un bucle, es decir, la instrucción **`interrumpir`** finaliza (termina) la ejecución de un bucle y, por tanto, el control del programa se transfiere (salta) a la primera instrucción después del bucle.

EJEMPLO Estúdiese el siguiente algoritmo:

```
algoritmo Numeros_opuestos_del_menos_10_al_mas_10

variables
   entero n, a

inicio
   a ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( n )
      si ( n = 0 )
         escribir( "ERROR: El cero no tiene opuesto." )
         interrumpir
         /* En el caso de que n sea un cero,
            el bucle se interrumpe. */
      fin_si
      escribir( "El opuesto es: ", -n )
      a ← a + n
   mientras ( n >= -10 y n <= 10 )
   escribir( "Suma: ", a )
fin
```

El algoritmo puede ser la solución del problema siguiente:

EJEMPLO Números opuestos (del -10 al 10) (Versión 1).

Diseñe el algoritmo de un programa que:

1º) Pida por teclado un número (dato entero).

2º) Si el número introducido por el usuario es distinto de cero, muestre por pantalla el mensaje:

- "El opuesto es: <*-número*>".

3º) Repita los pasos 1º y 2º, mientras que, el usuario introduzca un número mayor o igual que -10 y menor o igual que 10.

Pero, si el usuario introduce un cero, el bucle también finaliza, mostrándose por pantalla el mensaje:

- "ERROR: El cero no tiene opuesto.".

4º) Muestre por pantalla la suma de los números introducidos por el usuario.

En pantalla:

```
Introduzca un número entero: 15
El opuesto es: -15
Suma: 15
```


En este caso, el bucle ha finalizado porque la condición (**n >= -10 y n <= 10**) es **falsa**, ya que, **15** no es mayor o igual que **-10** y menor o igual que **10**.

Sin embargo, el bucle también puede finalizar, no porque sea **falsa** la condición (**n >= -10 y n <= 10**), sino, porque se ejecute la instrucción **interrumpir**. Esto ocurrirá cuando el usuario introduzca un cero. Por ejemplo:

```
Introduzca un número entero: 8
El opuesto es: -8
Introduzca un número entero: -7
El opuesto es: 7
Introduzca un número entero: 0
ERROR: El cero no tiene opuesto.
Suma: 1
```


Normalmente, cuando en un bucle se utiliza una instrucción **interrumpir**, la ejecución de esta se condiciona.

En el ejemplo, el bucle se interrumpe si la condición (**n = 0**) es **verdadera**. Nótese que, dicha condición no está contemplada en la condición de salida *estándar* del bucle, por lo que, a la condición (**n = 0**) se le considera **condición de salida interna** del bucle.

EJEMPLO No obstante, el problema también se puede resolver sin hacer uso de la instrucción **interrumpir**:

```
algoritmo Numeros_opuestos_del_menos_10_al_mas_10

 variables
    entero numero, acumulador

inicio
   acumulador ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( numero )
      si ( numero = 0 )
         escribir( "ERROR: El cero no tiene opuesto." )
      sino
         escribir( "El opuesto es: ", -numero )
         acumulador ← acumulador + numero
      fin_si
   mientras ( numero >= -10 y numero <= 10 y numero <> 0 )
   escribir( "Suma: ", acumulador )
fin
```

Obsérvese que, en este algoritmo, sí se contempla en la condición de salida del bucle la posibilidad de que el usuario teclee un cero, en cuyo caso, el bucle deja de iterar de forma *natural*.

Los resultados por pantalla de este algoritmo son idénticos a los del algoritmo anterior.

11.3. continuar

En pseudocódigo, para escribir una **instrucción de salto continuar** se utiliza la sintaxis:

```
continuar
```

La instrucción de salto **continuar** siempre se usa para interrumpir (romper) la ejecución normal de un bucle. Sin embargo, el control del programa no se transfiere a la primera instrucción después del bucle (como sí hace la instrucción **interrumpir**), es decir, el bucle no finaliza, sino que, finaliza la iteración en curso, transfiriéndose el control del programa a la condición de salida del bucle, para decidir si se debe realizar una nueva iteración o no.

Por tanto, la instrucción **continuar** finaliza (termina) la ejecución de una iteración de un bucle, pero, no la ejecución del bucle en sí. De forma que, la instrucción **continuar** salta (no ejecuta) las instrucciones que existan después de ella, en la iteración de un bucle.

EJEMPLO En el algoritmo siguiente se muestra cómo se puede utilizar la instrucción **continuar**:

```
algoritmo Numeros_opuestos_del_menos_10_al_mas_10

variables
   entero n, a

inicio
   a ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( n )
      si ( n = 0 )
         escribir( "ERROR: El cero no tiene opuesto." )
         continuar
         /* En el caso de que n sea un cero,
            la iteración en curso del bucle
            se interrumpe aquí. */
      fin_si
      escribir( "El opuesto es: ", -n )
      a ← a + n
   mientras ( n >= -10 y n <= 10 )
   escribir( "Suma: ", a )
fin
```

El código del algoritmo es el mismo que el del primer ejemplo visto en el apartado *“11.2. `interrumpir`”*, excepto por la instrucción **`interrumpir`**, que ha sido sustituida por la instrucción **`continuar`**. El algoritmo puede ser la solución para el problema siguiente, el cual se diferencia del ejemplo del apartado anterior en que si el usuario introduce un cero, el bucle no deja de iterar.

EJEMPLO Números opuestos (del -10 al 10) (Versión 2).

Diseñe el algoritmo de un programa que:

1º) Pida por teclado un número (dato entero).

2º) Si el número introducido por el usuario es distinto de cero, muestre por pantalla el mensaje:

- “El opuesto es: *<-número>*”.

En caso contrario, muestre el mensaje:

- “ERROR: El cero no tiene opuesto.”.

3º) Repita los pasos 1º y 2º, mientras que, el usuario introduzca un número mayor o igual que -10 y menor o igual que 10.

4º) Muestre por pantalla la suma de los números introducidos por el usuario.

En pantalla:

```
Introduzca un número entero: 2
El opuesto es: -2
Introduzca un número entero: 0
ERROR: El cero no tiene opuesto.
Introduzca un número entero: -59
El opuesto es: 59
Suma: -57
```


La instrucción **continuar** se ejecuta cuando el usuario introduce un cero, interrumpiendo la iteración en curso; pero, el bucle solamente finaliza cuando la condición (**n >= -10 y n <= 10**) sea **falsa**.

Normalmente, al igual que ocurre con la instrucción **interrumpir**, cuando en un bucle se utiliza una instrucción **continuar**, la ejecución de ésta también se condiciona.

En este último ejemplo, la iteración en curso del bucle se interrumpe si es **verdadera** la condición (**n = 0**).

EJEMPLO Ahora bien, el problema también se puede resolver sin hacer uso de la instrucción **continuar**:

```
algoritmo Numeros_opuestos_del_menos_10_al_mas_10

variables
   entero numero, acumulador

inicio
   acumulador ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( numero )
      si ( numero = 0 )
         escribir( "ERROR: El cero no tiene opuesto." )
      sino
         escribir( "El opuesto es: ", -numero )
         acumulador ← acumulador + numero
      fin_si
   mientras ( numero >= -10 y numero <= 10 )
   escribir( "Suma: ", acumulador )
fin
```

Los resultados por pantalla de este algoritmo son idénticos a los del algoritmo anterior.

11.4. `ir_a`

En pseudocódigo, para escribir una **instrucción de salto `ir_a`** se utiliza la sintaxis:

```
ir_a <nombre_de_la_etiqueta>
```

La instrucción de salto **`ir_a`** se puede usar en cualquier parte del cuerpo de un algoritmo, para transferir incondicionalmente el control del algoritmo (o programa) a la primera instrucción después de una etiqueta, o dicho de otra forma, al ejecutar una instrucción **`ir_a`**, el control del programa se transfiere (salta) a la primera instrucción después de una etiqueta. Una **etiqueta** se define mediante su nombre (identificador) seguido del carácter dos puntos (`:`).

EJEMPLO En el siguiente algoritmo se utiliza la instrucción **`ir_a`** para resolver el problema planteado en el apartado *"11.2. `interrumpir`"*:

```
algoritmo Numeros_opuestos_del_menos_10_al_mas_10

variables
   entero n, a

inicio
   a ← 0
   hacer
      escribir( "Introduzca un número entero: " )
      leer( n )
      si ( n = 0 )
         escribir( "ERROR: El cero no tiene opuesto." )
         ir_a etiqueta_1
         /* En el caso de que n sea un cero,
            el control del programa salta a la primera
            instrucción después de etiqueta_1. */
      fin_si
      escribir( "El opuesto es: ", -n )
      a ← a + n
   mientras ( n >= -10 y n <= 10 )

   etiqueta_1:
   escribir( "Suma: ", a )
fin
```

Los resultados por pantalla de este algoritmo son idénticos a los de los algoritmos del apartado *"11.2. `interrumpir`"*.

En pantalla:

```
Introduzca un número entero: -4
El opuesto es: 4
Introduzca un número entero: 12
El opuesto es: -12
Introduzca un número entero: 0
ERROR: El cero no tiene opuesto.
Suma: 8
```

{Abrirllave.com

Normalmente, al igual que ocurre con las instrucciones **interrumpir** y **continuar**, cuando en un algoritmo se utiliza una instrucción **ir_a**, la ejecución de ésta también se condiciona.

11.5. volver

En pseudocódigo, para escribir una **instrucción de salto volver** se utiliza la sintaxis:

```
volver <expresión>
```

Haremos uso de la instrucción **volver** cuando definamos subprogramas de tipo función, que estudiaremos en el apartado *“12.2.5. Funciones”*.

11.6. Ventajas de no usar las instrucciones de salto

Las ventajas de no usar las instrucciones de salto, especialmente la instrucción **ir_a**, se pueden resumir en:

- La legibilidad del algoritmo es mayor.
- La probabilidad de cometer errores en el algoritmo es menor.
- Es más fácil realizar cambios o corregir errores en el algoritmo.
- Nunca se altera (rompe) la secuencia de ejecución normal del programa. Dicho de otro modo, el programa siempre se ejecuta de un modo natural.

Capítulo 12

Llamadas a subalgoritmos

Como ya hemos estudiamos en el capítulo 6 "Instrucciones primitivas", en pseudocódigo, las instrucciones que se utilizan para diseñar algoritmos se pueden clasificar en:

- Primitivas (asignación, salida y entrada).
- De control (alternativas, repetitivas y de salto).
- Llamadas a subalgoritmos (llamadas a subprogramas).

Las instrucciones primitivas y de control ya han sido estudiadas. Así pues, solamente faltan por explicar las llamadas a subalgoritmos (subprogramas).

Un subalgoritmo se convertirá en un subprograma cuando se codifique en un lenguaje de programación específico.

Un **subprograma** es un programa, el cual, es llamado desde otro programa o subprograma. Por tanto, un subprograma solamente se ejecutará cuando sea llamado desde otro programa o subprograma.

12.1. Problemas y subproblemas

Utilizando el método "divide y vencerás", siempre que se pueda, es conveniente subdividir los problemas en otros más pequeños (subproblemas) y, en consecuencia, más fáciles de resolver.

EJEMPLO Un problema se puede segmentar en otros más pequeños:

- Subproblema 1
- Subproblema 2
- Subproblema 3

Además, si los subproblemas obtenidos siguen siendo demasiado grandes, de nuevo, puede ser conveniente que también estos sean fragmentados. Así pues, el subproblema 1 se puede subdividir en otros subproblemas:

- Subproblema 1.1
- Subproblema 1.2
- Subproblema 1.3

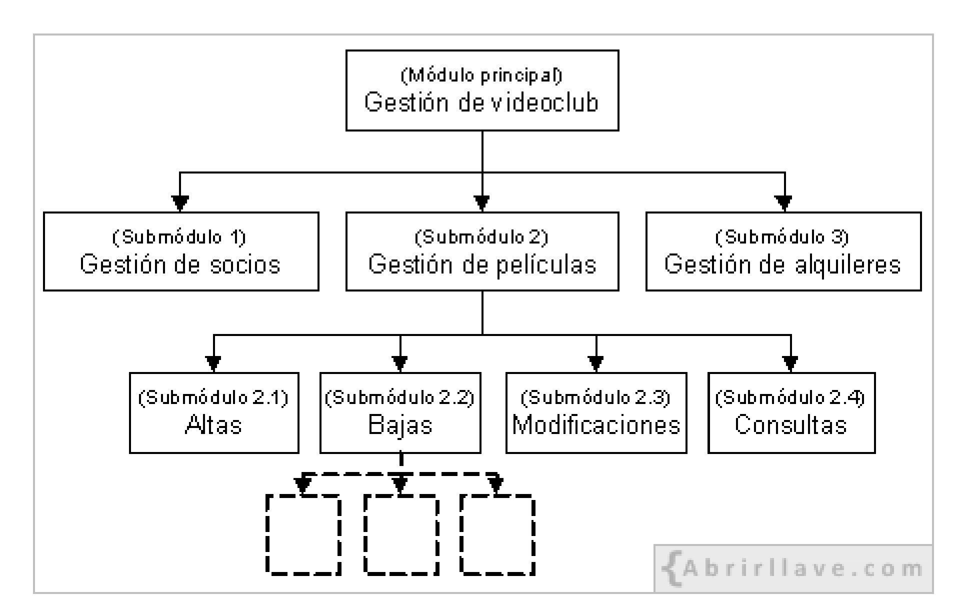
Y así sucesivamente. De forma que, por ejemplo, el subproblema 1.1 se puede fraccionar en otros todavía más pequeños:

- Subproblema 1.1.1
- Subproblema 1.1.2
- Subproblema 1.1.3

Etcétera.

12.2. Subprogramas

Como puede verse en el apartado de diseño de un programa del tutorial de programación de Abrirllave, el programa –módulo principal– que da solución a un problema, suele descomponerse en otros submódulos (subprogramas) más pequeños, que también se pueden fraccionar, y así sucesivamente.



Los subalgoritmos (subprogramas) se pueden clasificar en:

- Procedimientos.
- Funciones.

En este capítulo vamos a estudiar las diferencias y semejanzas existentes entre un procedimiento y una función.

12.2.1. Procedimientos

En pseudocódigo, la sintaxis que se utiliza para escribir un procedimiento es muy similar a la que se usa para escribir un algoritmo.

```
/* Cabecera */

procedimiento <nombre_del_procedimiento>(
[ <lista_de_parámetros_formales> ] )

/* Declaraciones */

[ constantes
     <declaraciones_de_constantes> ]
[ tipos_de_datos
     <declaraciones_de_tipos_de_datos> ]
[ variables
     <declaraciones_de_variables> ]

/* Cuerpo */

inicio
   <bloque_de_instrucciones>
fin
```

Existen dos diferencias importantes entre la sintaxis de un algoritmo y la de un procedimiento:

- En vez de la palabra reservada **algoritmo**, se debe escribir la palabra reservada **procedimiento**.
- En un procedimiento, después del **<nombre_del_procedimiento>**, se deben escribir los paréntesis "**()**", entre los cuales, opcionalmente, se pueden declarar parámetros formales.

Más adelante estudiaremos qué son y para qué sirven los parámetros formales. De momento, para entender cómo se puede hacer uso de los procedimientos –sin parámetros– estúdiese el siguiente problema.

EJEMPLO Menú de opciones.

Se quiere diseñar el algoritmo de un programa que:

1º) Muestre un menú con 4 opciones:

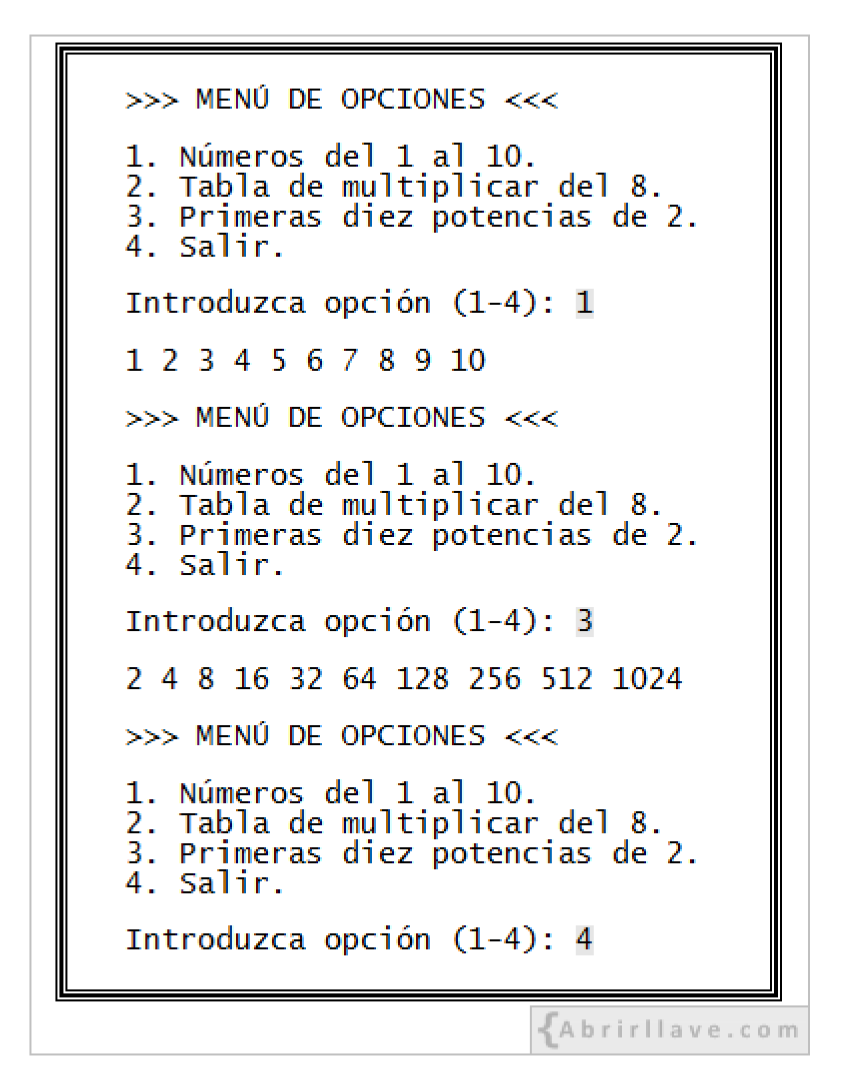
- Mostrar los números del 1 al 10 (ambos inclusive).
- Mostrar la tabla de multiplicar del 8.
- Mostrar las primeras diez potencias de 2.
- Salir.

2º) Pida por teclado la opción deseada (dato carácter). Deberá ser introducida, mientras que, no sea mayor o igual que '1' y menor o igual que '4'.

3º) Ejecute la opción del menú seleccionada.

4º) Repita los pasos 1º, 2º y 3º, mientras que, el usuario no seleccione la opción 4 (Salir) del menú.

En pantalla:



Sin usar subalgoritmos, la solución al problema puede ser la siguiente:

```
algoritmo Menu_de_opciones

variables
   caracter opcion
   entero numero, contador, resultado

inicio
   hacer
      escribir( ">>> MENÚ DE OPCIONES <<<" )
      escribir( "1. Números del 1 al 10." )
      escribir( "2. Tabla de multiplicar del 8." )
      escribir( "3. Primeras diez potencias de 2." )
      escribir( "4. Salir." )
      escribir( "Introduzca opción: " )

      /* Filtramos la opción elegida por el usuario. */

      hacer
         leer( opcion )
      mientras ( opcion < '1' o opcion > '4' )

      /* La opción solamente puede ser 1, 2, 3 ó 4. */

      segun_sea ( opcion )
         '1' : para numero ← 1 hasta 10 hacer
                  escribir( numero )
               fin_para
         '2' : para contador ← 1 hasta 10 hacer
                  contador ← contador * 8
                  escribir( "8 * ", contador, " = ",
                            resultado )
               fin_para
         '3' : para contador ← 1 hasta 10 hacer
                  escribir( 2 ** contador )
               fin_para
      fin_segun_sea
   mientras ( opcion <> '4' )
fin
```

En este caso, parece obvio que cada una de las opciones del menú puede considerarse como un subproblema:

- Subproblema 1: Mostrar los números del 1 al 10 (ambos inclusive).
- Subproblema 2: Mostrar la tabla de multiplicar del 8.
- Subproblema 3: Mostrar las primeras diez potencias de 2.

Los subalgoritmos (procedimientos) que dan solución a dichos subproblemas, pueden ser:

```
procedimiento Numeros_del_1_al_10()

variables
   entero numero

inicio
   para numero ← 1 hasta 10 hacer
      escribir( numero )
   fin_para
fin
```

```
procedimiento Tabla_de_multiplicar_del_8()

variables
   entero contador, resultado

inicio
   para contador ← 1 hasta 10 hacer
      resultado ← contador * 8
      escribir( "8 * ", contador, " = ", resultado )
   fin_para
fin
```

```
procedimiento Primeras_diez_pontencias_de_2()

variables
   entero contador

inicio
   para contador ← 1 hasta 10 hacer
      escribir( 2 ** contador )
   fin_para
fin
```

Además, al conjunto de instrucciones de salida que se utilizan para mostrar el menú por pantalla, también se le puede considerar como un subproblema:

- Subproblema 4: Mostrar menú por pantalla.

Para ello, el procedimiento propuesto es:

```
procedimiento Menu_por_pantalla()

inicio
   escribir( ">>> MENÚ DE OPCIONES <<<" )
   escribir( "1. Números del 1 al 10." )
   escribir( "2. Tabla de multiplicar del 8." )
   escribir( "3. Primeras diez potencias de 2." )
   escribir( "4. Salir." )
   escribir( "Introduzca opción: " )
fin
```

De manera que, la solución al problema planteado puede venir dada por un módulo principal (programa principal):

- **`Menu_de_opciones`**

Y cuatro submódulos (subprogramas):

- **`Numeros_del_1_al_10()`**
- **`Tabla_de_multiplicar_del_8()`**
- **`Primeras_diez_potencias_de_2()`**
- **`Menu_por_pantalla()`**

Por otra parte, para hacer una llamada a un procedimiento, la sintaxis es:

```
<nombre_del_procedimiento>( [ <lista_de_parámetros_actuales> ] )
```

En una llamada a un procedimiento se debe escribir el identificador de dicho procedimiento, seguido de los paréntesis "**()**", entre los cuales, opcionalmente, se puede indicar una lista de parámetros actuales. En el apartado *"12.2.4.2. Parámetros actuales"* vamos a estudiar qué son y para qué sirven los parámetros actuales.

Por tanto, usando los cuatro subalgoritmos (procedimientos) anteriores, la solución algorítmica al problema del ejemplo plantado en este apartado (Menú de opciones), puede ser la siguiente:

```
algoritmo Menu_de_opciones

variables
   caracter opcion

inicio
   hacer
      Menu_por_pantalla()    /* Llamada a subalgoritmo */

      /* Filtramos la opción elegida por el usuario. */

      hacer
         leer( opcion )
      mientras ( opcion < '1' o opcion > '4' )

      /* La opción solamente puede ser 1, 2, 3 ó 4. */

      segun_sea ( opcion )
         '1' : Numeros_del_1_al_10()
         '2' : Tabla_de_multiplicar_del_8()
         '3' : Primeras_diez_potencias_de_2()
      fin_segun_sea
   mientras ( opcion <> '4' )
fin
```

Cuando se hace una llamada a un procedimiento, este se ejecuta y, al finalizar, devuelve el control del flujo al programa (o subprograma) llamante, el cual continuará su ejecución con la siguiente instrucción a la llamada.

12.2.2. Declaraciones locales y globales

Las declaraciones de variables, constantes y tipos de datos realizadas en un subprograma (submódulo) se dice que son **declaraciones de ámbito local**, es decir, dichas variables, constantes y tipos de datos, solamente pueden utilizarse en dicho subprograma.

Por el contrario, las declaraciones realizadas en el programa (módulo) principal, se dice que son **declaraciones de ámbito global**, es decir, las variables, constantes y tipos de datos declarados en el programa principal, pueden utilizarse en el programa principal y en todos los subprogramas.

En los procedimientos propuestos para resolver el problema del ejemplo del apartado anterior *"12.2.1. Procedimientos"*, se han declarado las siguientes variables locales:

- **`numero`**, en el procedimiento **`Numeros_del_1_al_10`**.
- **`contador`** y **`resultado`**, en el procedimiento **`Tabla_de_multiplicar_del_8`**.
- **`contador`**, en el procedimiento **`Primeras_diez_potencias_de_2`**.

En el algoritmo **`Menu_por_pantalla`** no se ha realizado ninguna declaración de ámbito local.

Al hacer una llamada a un subprograma, es, en ese momento, y no antes, cuando se reserva espacio de memoria para las variables locales de ese subprograma llamado. Dicho espacio de memoria se libera cuando el subprograma finaliza su ejecución, devolviendo el control del flujo al programa (o subprograma) que lo llamó.

Por tanto, es importante entender que, la variable contador declarada en el procedimiento **`Tabla_de_multiplicar_del_8`**, no representa al mismo espacio de memoria que la variable **`contador`** declarada en el procedimiento **`Primeras_diez_potencias_de_2`**.

Igualmente, si se hubiese declarado otra variable **`contador`** en el programa principal, también esta representaría a un espacio de memoria diferente.

En el caso de que se declaren dos variables con el mismo identificador, una local (en un subprograma) y otra global (en el programa principal), al hacer referencia a dicho identificador, pueden ocurrir dos cosas:

- Si se hace en el programa principal, es obvio que se está referenciando a la variable global.
- Pero, si se hace en el subprograma, entonces se está referenciando a la variable local.

Asimismo, a las declaraciones de constantes y tipos de datos, se les aplica el mismo criterio.

EJEMPLO Partiendo de los dos procedimientos siguientes:

```
procedimiento Ejemplo_A()

variables
   entero n   /* Variable local */

inicio
   n ← 77
   escribir( n )
fin

procedimiento Ejemplo_B()

inicio
   escribir( n )
fin
```

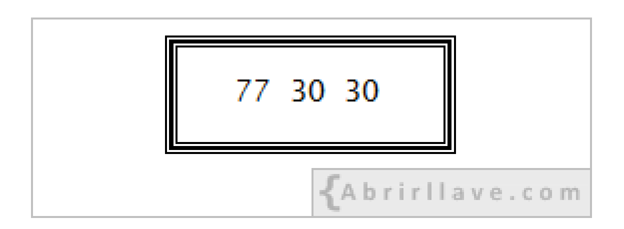
Podemos escribir el algoritmo:

```
algoritmo Ejemplo_C

variables
   entero n   /* Variable global */

inicio
   n ← 30
   Ejemplo_A()
   Ejemplo_B()
   escribir( n )
fin
```

La salida por pantalla es:



Obsérvese que, para que el procedimiento **Ejemplo_B** se ejecute correctamente, es necesario que la variable **n** esté declarada globalmente, y que además, esta contenga un dato (valor entero) en el momento de la llamada al procedimiento **Ejemplo_B**. En cualquier caso, para que no se dé esta dependencia, es recomendable declarar dicha variable localmente, como se ha hecho en el procedimiento **Ejemplo_A**.

Del mismo modo, se pueden declarar dos variables con el mismo identificador, una local (en un subprograma) y otra también local (en otro subprograma llamado por el anterior). En cuyo caso, al hacer referencia a dicho identificador, se estará referenciando a la variable más local.

Las declaraciones deben ser lo más locales posibles. Así que, cuantas menos declaraciones globales existan, mejor. Además, con el fin de evitar confusiones, y prevenir errores no deseados, es recomendable no usar el mismo identificador para una variable local, que para otra global.

12.2.3. Declaraciones de subprogramas

En pseudocódigo, los subprogramas también se pueden declarar locales o globales. Por consiguiente, tanto a la sección de declaraciones de un algoritmo (programa principal), como a la de un subalgoritmo (subprograma), se debe añadir la sintaxis:

```
[ subalgoritmos
     <declaraciones_de_subalgoritmos> ]
```

Para declarar un procedimiento –sin parámetros– se utiliza el identificador de dicho procedimiento, seguido de los paréntesis "**()**". La sintaxis es:

```
<nombre_del_procedimiento>()
```

Por tanto, si al algoritmo del ejemplo del apartado *"12.2.1. Procedimientos"* le añadimos las declaraciones de los procedimientos que en él se utilizan, obtendremos el algoritmo:

```
algoritmo Menu_de_opciones

subalgoritmos
    /* Procedimientos globales */
    Numeros_del_1_al_10()
    Tabla_de_multiplicar_del_8()
    Primeras_diez_potencias_de_2()
    Menu_por_pantalla()

variables
    caracter opcion   /* Variable global */

inicio
   hacer
      Menu_por_pantalla()   /* Llamada a subalgoritmo */

      hacer
         leer( opcion )
      mientras ( opcion < '1' o opcion > '4' )

      segun_sea ( opcion )
         '1' : Numeros_del_1_al_10()
         '2' : Tabla_de_multiplicar_del_8()
         '3' : Primeras_diez_potencias_de_2()
      fin_segun_sea
   mientras ( opcion <> '4' )
fin
```

Haciendo lo mismo con el algoritmo del ejemplo del apartado *"12.2.2. Declaraciones locales y globales"*, obtendremos el algoritmo:

```
algoritmo Ejemplo_C

subalgoritmos
   /* Procedimientos globales */
   Ejemplo_A()
   Ejemplo_B()

variables
   entero n   /* Variable global */

inicio
   n ← 30
   Ejemplo_A()
   Ejemplo_B()
   escribir( n )
fin
```

EJEMPLO Estúdiese ahora el procedimiento siguiente:

```
procedimiento Ejemplo_A()

subalgoritmos
   Ejemplo_B()   /* Procedimiento local */

variables
   entero n   /* Variable local */

inicio
   Ejemplo_B()   /* Llamada a subalgoritmo */
   n ← 26
   escribir( n )
fin
```

En el ejemplo, se ha declarado el procedimiento **Ejemplo_B** localmente. Por tanto, este únicamente puede ser llamado desde el procedimiento **Ejemplo_A**. Esto es posible especificarlo en muchos lenguajes estructurados, pero no en otros. Por ejemplo, en C esta característica no se cumple, ya que, no se puede definir un subprograma dentro de otro. Por tanto, cuando se escribe un programa en C, todos los subprogramas se declaran globalmente.

Y dado el procedimiento:

```
procedimiento Ejemplo_B()

variables
   entero n   /* Variable local */

inicio
   n ← 90
   escribir( n )
fin
```

Y el algoritmo:

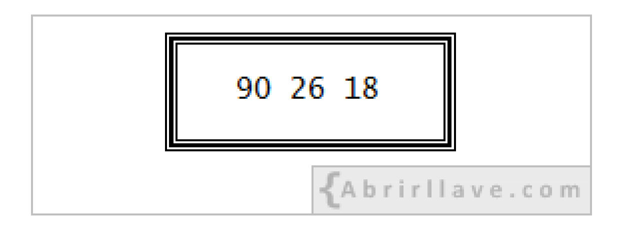
```
algoritmo Ejemplo_D

subalgoritmos
   Ejemplo_A()    /* Procedimiento global */

variables
   entero n   /* Variable global */

inicio
   n ← 18
   Ejemplo_A()    /* Llamada a subalgoritmo */
   escribir( n )
fin
```

Su salida por pantalla es:



Al igual que con las variables, constantes y tipos de datos, también es conveniente declarar los subalgoritmos lo más localmente posible. Así como, no usar el mismo identificador para un subalgoritmo local, que para otro global.

Por otro lado, como se ha podido ver, el espacio de memoria reservado para las variables globales –en este caso una variable, **n**– no se libera hasta que el programa principal finaliza su ejecución, a diferencia de las variables locales.

12.2.4. Parámetros

Los **parámetros** –también llamados argumentos– se emplean, opcionalmente, para transferir datos de un programa (o subprograma) llamante, a otro llamado, y viceversa (del llamado al llamante), o dicho de otro modo, en una llamada a un subprograma, el llamante y el llamado se pueden enviar datos entre sí, mediante parámetros. De manera que, en una llamada a un subprograma, los parámetros se usan para:

- Proporcionar uno o más datos de entrada al llamado.
- Devolver uno o más datos de salida al llamante.

Por tanto, los parámetros se pueden clasificar en:

- De entrada.
- De salida.
- Y también, de entrada y salida.

Como su propio nombre indica, un **parámetro de entrada y salida** se utiliza para proporcionar un dato de entrada al llamado y, también, para devolver un dato de salida al llamante.

Para entender cómo se puede hacer uso de los parámetros, estúdiese el siguiente problema.

EJEMPLO Multiplicación de dos números enteros.

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado dos números (datos enteros).

2º) Calcule la multiplicación de los dos números introducidos por el usuario.

3º) Muestre por pantalla el resultado (dato entero).

En pantalla:

```
Introduzca el primer número: 26
Introduzca el segundo número: 3
La multiplicación es: 78
```


Sin usar subalgoritmos, la solución al problema puede ser la siguiente:

```
algoritmo Multiplicacion_de_dos_numeros_enteros

variables
   entero a, b, r

inicio
   escribir( "Introduzca el primer número: " )
   leer( a )
   escribir( "Introduzca el segundo número: " )
   leer( b )
   r ← a * b
   escribir( "La multiplicación es: ", r )
fin
```

En este caso, vamos a considerar como un subproblema la realización de la multiplicación.

- Subproblema: Realizar la multiplicación de dos números enteros.

Recuérdese que, en pseudocódigo, la sintaxis que se utiliza para escribir la cabecera de un procedimiento es:

```
procedimiento <nombre_del_procedimiento>(
[ <lista_de_parámetros_formales> ] )
```

12.2.4.1. Parámetros formales

Los **parámetros formales** –también llamados ficticios– son variables locales que se declaran en la cabecera de un procedimiento, en las cuales se almacenarán:

- Los datos de entrada que se le proporcionen al procedimiento en la llamada.
- Así como, los datos de salida que se devolverán al subprograma llamante.
- Y también, los datos de entrada y salida.

En un subprograma, las variables locales declaradas en la sección de declaraciones, se diferencian de las variables locales declaradas en la cabecera (parámetros), en que, estas últimas, se utilizan para transferir datos entre el llamante y el llamado, y las otras no.

Para escribir la **`<lista_de_parámetros_formales>`** de un subalgoritmo, la sintaxis es:

```
[ E <nombre_del_tipo_de_dato_1> : <lista_de_variables_1> ;
  E <nombre_del_tipo_de_dato_2> : <lista_de_variables_2> ;
  ...
  E <nombre_del_tipo_de_dato_n> : <lista_de_variables_n> ; ]

[ S <nombre_del_tipo_de_dato_1> : <lista_de_variables_n+1> ;
  S <nombre_del_tipo_de_dato_2> : <lista_de_variables_n+2> ;
  ...
  S <nombre_del_tipo_de_dato_n> : <lista_de_variables_m> ; ]

[ E/S <nombre_del_tipo_de_dato_1> : <lista_de_variables_m+1> ;
  E/S <nombre_del_tipo_de_dato_2> : <lista_de_variables_m+2> ;
  ...
  E/S <nombre_del_tipo_de_dato_n> : <lista_de_variables_p> ]
```

En donde, **`E`**, **`S`** y **`E/S`**, indican que los parámetros siguientes son de entrada, de salida y de entrada/salida, respectivamente.

El carácter punto y coma (`;`) únicamente se debe poner, si después de una lista de variables, hay otra.

Las variables de una lista de variables deben ir separadas por el carácter coma (`,`).

De modo que, suponiendo que al subalgoritmo (procedimiento) que da solución al subproblema planteado en el apartado anterior –realizar la multiplicación de dos números enteros– se le pasen dos datos de entrada (los dos números introducidos por el usuario) en la llamada, y devuelva un dato de salida (el resultado de la multiplicación), el procedimiento que da solución a dicho subproblema, puede ser:

```
procedimiento Multiplicar(
   E entero n1, n2;
   S entero resultado )

inicio
   resultado ← n1 * n2
fin
```

Los parámetros **n1** y **n2**, son variables de entrada (**E**). Por tanto, cuando se realice una llamada al procedimiento **Multiplicar**, se tienen que proporcionar los datos que se almacenarán –recogerán– en dichos parámetros.

Por otra parte, cuando el procedimiento **Multiplicar** finalice, el parámetro **resultado** debe contener un dato que se devolverá al llamante. Dicho dato, se almacenará en una variable local declarada en el llamante, o en una global.

12.2.4.2. Parámetros actuales

Recuérdese que, en pseudocódigo, la sintaxis que se utiliza para hacer una llamada a un procedimiento es:

```
<nombre_del_procedimiento>( [ <lista_de_parámetros_actuales> ] )
```

Los parámetros de una **<lista_de_parámetros_actuales>** deben ir separados por el carácter coma (**,**).

En una llamada a un subprograma, el número de **parámetros actuales** –también llamados reales– debe coincidir con el número de parámetros formales declarados en el subprograma, existiendo una correspondencia de tipos de datos entre ellos, es decir, el primer parámetro formal debe ser del mismo tipo de dato que el primer parámetro actual, y así con todos.

Los parámetros actuales que se correspondan con parámetros formales de entrada, pueden ser expresiones. De esta forma, el resultado de evaluar un parámetro actual (expresión), se proporciona como dato de entrada al llamado. Sin embargo, los parámetros actuales que se correspondan con parámetros formales de salida o de entrada y salida, únicamente pueden ser variables, ya que, un dato de salida devuelto por el llamante, se almacena en un parámetro actual, el cual, obviamente, solamente puede ser una variable.

Por otro lado, para declarar un procedimiento –con parámetros– se utiliza el identificador de dicho procedimiento, seguido de los paréntesis "**()**", entre los cuales, se deben escribir los tipos de datos de los parámetros formales que se hayan declarado en el procedimiento. La sintaxis es:

```
<nombre_del_procedimiento>( <lista_de_tipos_de_datos> )
```

Para escribir la **<lista_de_tipos_de_datos>** en la declaración de un subalgoritmo, la sintaxis es:

```
[ E <lista_de_tipos_de_datos_de_entrada> ; ]
[ S <lista_de_tipos_de_datos_de_salida> ; ]
[ E/S <lista_de_tipos_de_datos_de_entrada_y_salida> ]
```

El carácter punto y coma (**;**) únicamente se debe poner, si después de una lista de tipos de datos, hay otra.

Los tipos de datos de una lista de tipos de datos deben ir separados por el carácter coma (**,**).

Así pues, usando el procedimiento **Multiplicar**, la solución algorítmica al problema del ejemplo (Multiplicación de dos números enteros) del apartado *"12.2.4. Parámetros"*, puede ser la siguiente:

```
algoritmo Multiplicacion_de_dos_numeros_enteros

subalgoritmos
   Multiplicar( E entero, entero;
                S entero)

variables
   entero a, b, r

inicio
   escribir( "Introduzca el primer número: " )
   leer( a )
   escribir( "Introduzca el segundo número: " )
   leer( b )
   Multiplicar( a, b, r )
   escribir( "La multiplicación es: ", r )
fin
```

Suponiendo, por ejemplo, que el usuario desee calcular la multiplicación de los números 5 y 8, en pantalla se mostrará:

```
Introduzca el primer número: 5

Introduzca el segundo número: 8

La multiplicación es: 40
```


Por tanto, cuando en el algoritmo se hace la llamada al procedimiento **Multiplicar**:

```
Multiplicar( a, b, r )
```

En los parámetros **n1** y **n2** del procedimiento, se almacenan –copian– los datos de entrada (**5** y **8**) proporcionados en la llamada.

5 y **8** son los resultados de evaluar las expresiones (variables en este caso) **a** y **b**, respectivamente.

12.2.4.3. Paso por valor

Usando como ejemplo el algoritmo **Multiplicacion_de_dos_numeros_enteros** del apartado anterior *"12.2.4.2. Parámetros Actuales"*:

```
algoritmo Multiplicacion_de_dos_numeros_enteros

subalgoritmos
   Multiplicar( E entero, entero;
                S entero)

variables
   entero a, b, r

inicio
   escribir( "Introduzca el primer número: " )
   leer( a )
   escribir( "Introduzca el segundo número: " )
   leer( b )
   Multiplicar( a, b, r )
   escribir( "La multiplicación es: ", r )
fin
```

Y siendo el código del procedimiento **Multiplicar** el siguiente:

```
procedimiento Multiplicar(
   E entero n1, n2;
   S entero resultado )

inicio
   resultado ← n1 * n2
fin
```

Cuando el valor (dato) de un parámetro actual (**a** por ejemplo), se transfiere –copia– a un parámetro formal de entrada (**n1**, en este caso), se dice que se está realizando un paso por valor.

El paso por valor implica la asignación:

```
<parámetro_formal_de_entrada> ← <parámetro_actual>
```

De modo que, en el algoritmo **Multiplicacion_de_dos_numeros_enteros**, se producen los pasos por valor siguientes:

```
n1 ← a
n2 ← b
```

12.2.4.4. Paso por referencia

Usando como ejemplo el algoritmo **Multiplicacion_de_dos_numeros_enteros** del apartado *"12.2.4.2. Parámetros Actuales"*:

```
algoritmo Multiplicacion_de_dos_numeros_enteros

subalgoritmos
   Multiplicar( E entero, entero;
                S entero)

variables
   entero a, b, r

inicio
   escribir( "Introduzca el primer número: " )
   leer( a )
   escribir( "Introduzca el segundo número: " )
   leer( b )
   Multiplicar( a, b, r )
   escribir( "La multiplicación es: ", r )
fin
```

Siendo el código del procedimiento **Multiplicar** el siguiente:

```
procedimiento Multiplicar(
   E entero n1, n2;
   S entero resultado )

inicio
   resultado ← n1 * n2
fin
```

Y suponiendo que el usuario calcule la multiplicación de los números 5 y 8, mostrándose en pantalla:

```
Introduzca el primer número: 5

Introduzca el segundo número: 8

La multiplicación es: 40
```

{Abrirllave.com

En el parámetro **r** (variable de salida) se almacenará el valor (dato) **40**, como consecuencia de la instrucción de asignación:

```
resultado ← n1 * n2
```

Puede pensarse que, cuando finalice el procedimiento **Multiplicar**, se efectuará la asignación:

```
r ← resultado
```

Pero, en realidad, **resultado** no es una variable que almacene un dato de tipo entero, ya que, un parámetro formal de salida, como es el caso de **resultado**, representa al espacio de memoria en el cual se almacena la dirección de memoria del parámetro actual correspondiente, **r** en este caso.

Por tanto, cuando al parámetro formal de salida (**resultado**) se le asigna un valor dentro del procedimiento **Multiplicar**. Lo que se está haciendo realmente, es asignar dicho valor al parámetro actual correspondiente (**r**), es decir, **resultado** hace referencia a **r** y, por tanto, se dice entonces que se está realizando un paso por referencia.

Es importante comprender que, cuando se realiza un paso por valor, si se modifica el valor del parámetro formal en el subprograma llamado, haciendo por ejemplo:

```
n1 ← 3
```

Dicha modificación no afectaría al parámetro actual (**a** en este caso), que seguiría conteniendo el valor **5**. Pero, si el paso fuese por referencia, entonces sí que afectaría.

12.2.5. Funciones

En pseudocódigo, la sintaxis que se utiliza para escribir una función es muy similar a la que se usa para escribir un procedimiento:

```
/* Cabecera */

<tipo_de_dato> funcion <nombre_de_la_función>(
[ <lista_de_parámetros_formales> ] )

/* Declaraciones */

[ constantes
      <declaraciones_de_constantes> ]
[ tipos_de_datos
      <declaraciones_de_tipos_de_datos> ]
[ variables
      <declaraciones_de_variables> ]

/* Cuerpo */

inicio
   <bloque_de_instrucciones>
   volver <expresión>
fin
```

Existen dos diferencias importantes entre la sintaxis de un procedimiento y de una función:

1. En vez de la palabra reservada **procedimiento**, se debe escribir la palabra reservada **funcion**.
2. Una función devuelve siempre un valor. Por tanto, en una función siempre debe indicarse el **<tipo_de_dato>** del valor que devuelve la función, y el valor en sí, mediante la instrucción **volver <expresión>**.

De modo que, si queremos realizar la multiplicación de dos números enteros por medio de una función, podemos escribir, por ejemplo:

```
entero funcion Multiplicar(
    E entero n1, n2 )

variables
   entero resultado

inicio
   resultado ← n1 * n2
   volver resultado
fin
```

O también:

```
entero funcion Multiplicar(
    E entero n1, n2 )

inicio
   volver n1 * n2
fin
```

Por otro lado, para declarar una función en un algoritmo se utiliza la sintaxis:

```
<tipo_de_dato> <nombre_de_la_función>( <lista_de_tipos_de_datos> )
```

Y, para hacer una llamada a una función, la sintaxis es:

```
<nombre_de_la_función>( [ <lista_de_parámetros_actuales> ] )
```

En consecuencia, usando la función **Multiplicar**, la solución algorítmica al problema del ejemplo (Multiplicación de dos números enteros) del apartado *"12.2.4. Parámetros"*, puede ser la siguiente:

```
algoritmo Multiplicacion_de_dos_numeros_enteros

subalgoritmos
   entero Multiplicar( E entero, entero )

variables
   entero a, b

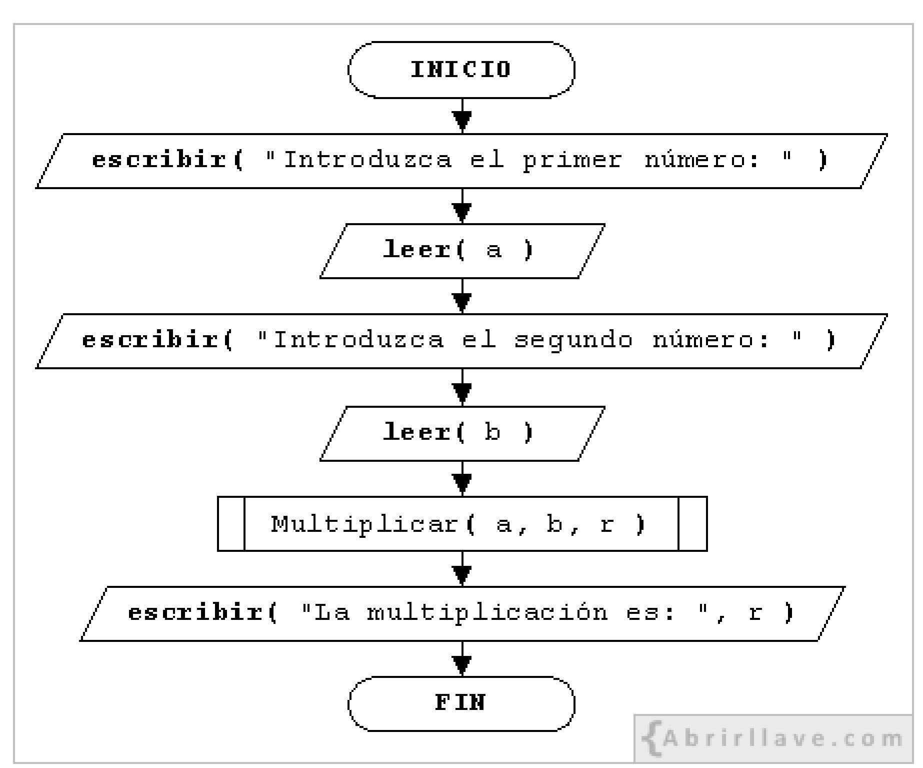
inicio
   escribir( "Introduzca el primer número: " )
   leer( a )
   escribir( "Introduzca el segundo número: " )
   leer( b )
   escribir( "La multiplicación es: ", Multiplicar( a, b ) )
fin
```

12.2.6. Representación mediante diagramas de flujo

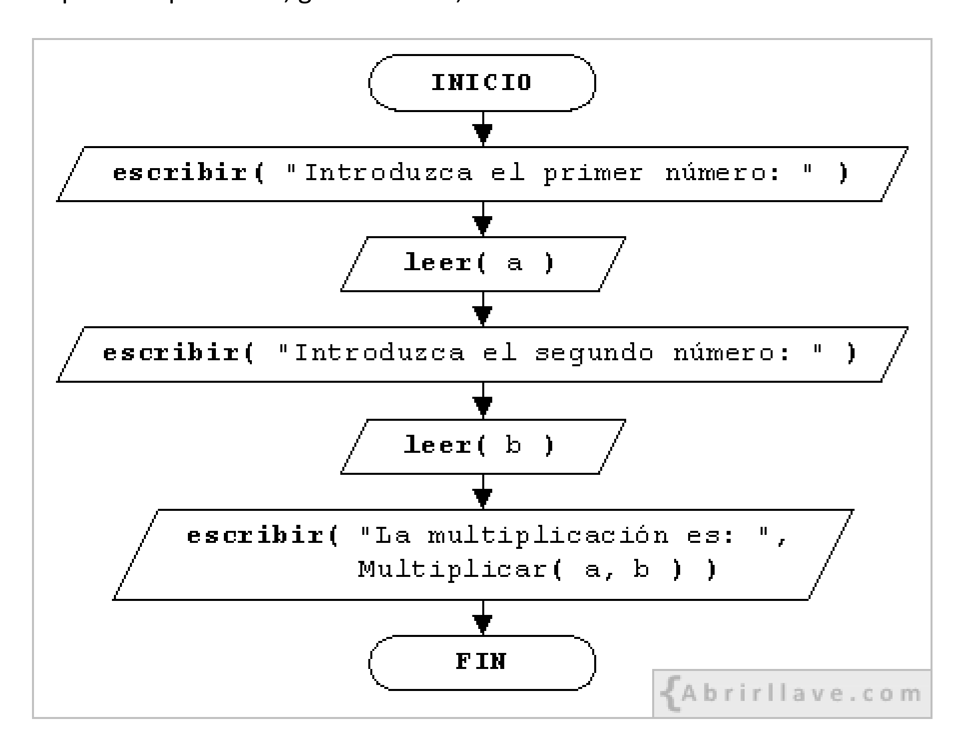
En un ordinograma, para representar una llamada a un subprograma se utiliza el símbolo:



De forma que, por ejemplo, el algoritmo **Multiplicacion_de_dos_numeros_enteros** propuesto en el apartado *"11.2.4.2 Parámetros actuales"*, se puede representar, de manera gráfica, de la siguiente forma:



Ahora bien, cuando se trata de una llamada a una función, en vez de a un procedimiento, se puede prescindir del símbolo de llamada a un subprograma. En consecuencia, el algoritmo **`Multiplicacion_de_dos_numeros_enteros`** visto en el apartado *“12.2.5. Funciones”* se puede representar, gráficamente, como se muestra a continuación:



12.3. Recursividad

En programación, cuando desde un subprograma se realiza una llamada a sí mismo, se dice que se está haciendo uso de la **recursividad**. Por tanto, un subprograma recursivo es aquel que se llama a sí mismo. No obstante, hay que saber, que toda solución recursiva tiene su correspondiente solución iterativa.

EJEMPLO Factorial de un número.

Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado un número (dato entero).

2º) Si el número introducido no es mayor que cero, muestre por pantalla el mensaje:

- “ERROR: El número debe ser mayor que cero.”

En caso contrario, muestre el factorial del número introducido por el usuario.

Nota: El factorial de un número natural (n!) es el producto de todos los números que hay entre el uno (1) y dicho número (n), ambos inclusive. Por ejemplo, el factorial de 4 es el resultado de multiplicar 1 * 2 * 3 * 4, es decir, 4! = 24.

En pantalla:

```
Introduzca un número entero: -4
ERROR: El número debe ser mayor que cero.
Introduzca un número entero: 4
4! = 24
```


La solución al problema puede venir dada por el siguiente algoritmo:

```
algoritmo Factorial_de_un_numero

variables
   entero n

inicio
   escribir( "Introduzca un número: " )
   leer( n )
   si( n > 0 )
      escribir( n, "! = ", Factorial( n ) )
   sino
      escribir( "ERROR: El número debe ser mayor que cero." )
   fin_si
fin
```

Sin emplear recursividad, se puede escribir la función **Factorial** de la siguiente forma:

```
entero funcion Factorial(
   E entero numero )

variable
   entero contador, resultado

inicio
   resultado ← numero

   mientras ( numero > 2 )
      numero ← numero - 1
      resultado ← resultado * numero
   fin_mientras

   volver resultado
fin
```

Y, recursivamente:

```
entero funcion Factorial(
   E entero numero )

variable
   entero resultado

inicio
   si ( numero <> 1 )
      resultado ← Factorial( numero - 1 ) * numero
   sino
      resultado ← 1
   fin_si

   volver resultado
fin
```

En ambas soluciones, el resultado devuelto en la llamada a **Factorial** realizada en el algoritmo **Factorial_de_un_numero** es el mismo. Sin embargo, obsérvese que, en la segunda solución de la función **Factorial**, se llamará de forma recursiva al subprograma.

En cuanto a qué solución es mejor, la iterativa o la recursiva, hay que decir que, la solución iterativa suele utilizar menos memoria, no corre el riesgo de agotarla e, incluso, puede ser más rápida algorítmicamente. Ahora bien, en algunas ocasiones, la solución recursiva es más elegante y fácil de obtener, por lo que, dependiendo del problema, el programador debe decidir qué solución implementar.

EPÍLOGO

"Abrirllave.com" es un sitio web de tutoriales de informática y otros recursos relacionados.

Tutoriales de informática

En Abrirllave puede consultar, entre otros, los siguientes tutoriales:

- ***Tutorial de CMD***
  http://www.abrirllave.com/cmd/
- ***Tutorial de Desarrollo Web***
  http://www.abrirllave.com/desarrollo-web/
- ***Tutorial de Google Sites***
  http://www.abrirllave.com/google-sites/
- ***Tutorial de lenguaje C***
  http://www.abrirllave.com/c/
- ***Tutorial de SEO (Search Engine Optimization)***
  http://www.abrirllave.com/seo/

En la dirección web www.abrirllave.com/tutoriales.php puede consultar el listado de todos los tutoriales publicados en Abrirllave a día de hoy.

Abrirllave en redes sociales

Para mantenerse informado de la incorporación de nuevos contenidos a "Abrirllave.com", puede hacerlo a través de:

- ***Facebook***
  https://www.facebook.com/Abrirllave
- ***Google+***
  https://plus.google.com/+Abrirllavecom
- ***Slideshare***
  https://www.slideshare.net/abrirllave
- ***Twitter***
  https://twitter.com/Abrirllave

Publicar en Abrirllave

Si lo desea, en la página www.abrirllave.com/publicar.php se explica cómo puede compartir sus conocimientos de informática en "Abrirllave.com".